Compositional Reasoning About Aspects Using Alternating-time Logic

Benet Devereux
Department of Computer Science
University of Toronto
10 King's College Road
Toronto, Ontario, Canada
M5S 3G4
benet@cs.toronto.edu

ABSTRACT

Aspect-oriented programming offers greater modularity to the programmer, but it is not yet clear how best to reason about an aspect-oriented program in a modular way. We propose a translation of aspect-oriented programs into alternating transition systems (ATSs), which provide a decidable formal specification language, alternating-time logic, that allows us to specify which component is responsible for enforcing certain properties. We develop rules for compositional reasoning using these translations.

1. INTRODUCTION

It is an observed problem [5] with aspect-oriented programming that, while aspects do provide additional modularity of development, it is not yet well established how to reason about aspect-based programs in a modular way. Such compositional reasoning allows aspects to be not only designed in isolation, but also formally verified. Such formal verification is useful for tractably proving correctness of large programs, since the task can be decomposed, and also for promoting re-use of aspects.

Compositional reasoning in concurrent programming is a well-understood (though difficult) problem. Modules of concurrent programs can be verified separately through *assume-guarantee* reasoning [7]. The question we pose at the outset is: how much of this knowledge can be reused for compositional reasoning about aspects?

To begin answering this question, we propose a semantics of aspect-oriented programs based on *alternating transition systems* (ATSs), a variety of state-machine model which explicitly represents how multiple components work together to change the system's state, each only having partial control. In our model, we treat aspects as concurrent components which have the authority, at certain points, to take control and modify program state, possibly returning at a different point. The possible points of return are constrained, but allow for an aspect either to return control where it took it, or to skip a statement. This semantics is a direct translation from code into a low-level state machine model, but it should be the same as a code-weaver-based semantics [8], where the aspect code is woven into the source code, and then translated; with the additional information of the allocation of responsibilities to components (system and aspect). We are not proposing any new constructs for aspect languages, but rather an approach to automated compositional analysis of existing languages.

Compositional reasoning for concurrent systems often proceeds as follows: there are two communicating components, P and Q. We show first that P placed in composition with a suitable abstraction of Q is correct; then, that Q in composition with an abstraction of P is also correct. In this way, we avoid constructing the full state-space of $P \parallel Q$, which may not be possible in the available memory; we pay the cost of having to construct and verify the abstractions.

We argue that this model is general enough to allow us to represent complicated interactions of components to determine state transitions, and yet remain amenable to the existing methods of analysis of alternating transition systems, such as model-checking [1] and refinement checking [2]; as well as any new techniques which aspect verification may make necessary.

The contribution of this paper is a translation from a simple aspect language into the ATS formalism, which allows assume-guarantee reasoning, and a discussion of how this translation may be used to show that an aspect modifies a program correctly. The aspect language is similar to the fragment of AspectJ [8] which only deals with advice to running code, and not with modifications of the class hierarchy. We explain the proposed technique on an example, giving definitions as needed, and suggest two compositional proof rules for analysis of aspect-oriented programs.

2. EXAMPLE

As an example aspect, we take precondition-checking. A program module makes use of a `Point` class; some class

(a)

```
p0:  while (!button);
p1:  pt.movePoint(x,y)
p2:  goto p0
```

(b)

```
pointcut mp(x, y):
 call (Point.movePoint)
   && args(x,y)
-----------------------
       over mp(x,y):
a1:    if (x>=0, y>=0)
a2:      continue
a3:    else
a4:      skipover
```

Figure 1: (a): code using `movePoint`, (b): code for the precondition-checking aspect.

methods have preconditions, and if they are called without the precondition holding, their behavior is undefined. We take `movePoint` as one such method, following [5]; its precondition is that the coordinates given are non-negative. An aspect is defined which inserts checks of this precondition before any call to `movePoint`. The aspect skips over the call if the precondition is not met. It is assumed that `movePoint` is provably correct: that is, if it is invoked under the right conditions, it terminates with its postconditions satisfied.

The program is shown in Figure 1(a). Its user interface contains a canvas, two numeric text fields in which the user can fill in x and y coordinates, and a button to move an image to the specified location on the canvas. The program only ever reads the text fields, it does not write to them; and so their contents are determined by the environment alone. All other state is determined by the program. For this aspect to be correct, the combination of the aspect with the program must have the following properties:

- if the program calls `movePoint` with the proper precondition, the postcondition will be satisfied at the first program point after the invocation of `movePoint`; this was true before aspect imposition, and should *still* be true after the aspect is added
- the aspect should prevent `movePoint` from being called with bad parameters

Additionally, the problem of aspect interaction should also be addressed: there may be other aspects which run before, or after, or even during the precondition-guarantee aspect. The problem of formally representing the assumptions on *other* aspects under which the aspect continues to behave as specified seems similar to the problem of representing the system assumptions under which the aspect of interest behaves properly, but considerably more subtle. Though interaction of aspects is an important consideration, this preliminary work does not as yet deal with it.

3. ALTERNATING TRANSITION SYSTEM SEMANTICS

In this section, we give a sketch of the proposed semantics, using the `movePoint` example to illustrate it. The semantics is in most respects a standard operational semantics for an imperative language [9], only enriched with information about how individual agents (the system, the environment, and the aspect) control the state transitions. We refer the reader to the stated references for a formal treatment, and illustrate all definitions using the example.

Informally, an alternating transition system is a state machine where multiple agents each have partial control over the transition relation. Thus, in a single state, each agent may not be able to definitively choose a successor, but rather a set of possible successors: the actual state chosen from that set is contingent upon how the other agents choose. Thus, in a sense, the agents play a game to control the behaviour of the system. We say that an agent has a *capability* if it has a strategy to keep the behaviour within a certain set where every possible execution has some desired property; these capabilities will be expressed in *alternating-time logic*, to be described more in Section 4.

Definition 3.1 (Alternating Transition Systems). *An alternating transition system is a tuple $A = (\Omega, S, P, L, R)$ where:*

- *Ω is a finite set of agents;*
- *S is a set of states;*
- *P is a set of atomic propositions;*
- *$L : S \to 2^P$ is a function labelling states with sets of atomic propositions;*
- *$R : S \times \Omega \to 2^{2^S}$ is the transition relation;*

At each state s, every agent $a \in \Omega$ chooses one set of possible successors $T_a \in R(s,a)$; the intersection $\bigcap_{a \in \Omega} T_a$ must be a singleton, which is the chosen successor.

We look again at the program-aspect composition of Figure 1. At any point in execution, the environment has partial control over the evolution of system state: it may change the text fields and the button, but update of the program counter and the canvas is up to the system. Thus, when the environment moves at a state, it chooses a set of successors: for instance, it can choose the set of all states where x is 5 and y is 3 as successors, but it cannot choose one uniquely, because the system has control over the program counter and the canvas. Symmetrically, the system chooses a set determined by its choices; the intersection of these two (along with the aspect's actions) produces a unique successor.

To represent ATSs visually, we use state diagrams annotated with decision nodes between states. The states are labelled circles, and the decision nodes are small squares with the name of the agent making the decision. Note that the decision points between states are arranged in a sequence for the sake of visual clarity, but that in fact all decisions are made *simultaneously* by the agents, none having knowledge of what the other agents are choosing. A fragment of the ATS translation for the system/aspect composition of Figure 1 is shown in Figure 3. Solid circles are states where the program is active, dashed circles are states where the aspect is active. A starred transition between two states indicates that it is reachable by collaboration of all three agents.

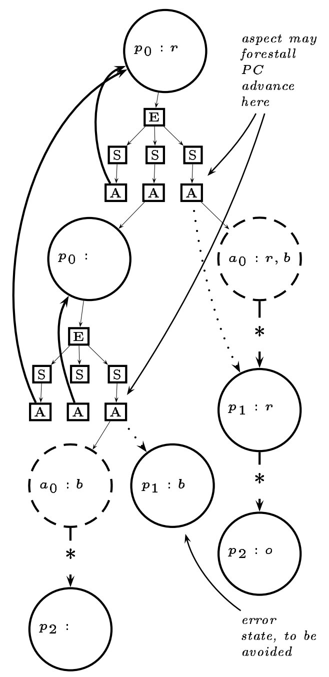

Figure 2: Fragment of the ATS translation

The states are labelled with the program point (either in the program or the aspect: while one is active, the other is assumed to be suspended), followed by a list of the propositional variables holding in that state. These are abstractions of the concrete program state, defined as follows:

Abstract Variable	Description	Definition
r	Precondition holds	$(\mathtt{x} \geq 0) \wedge (\mathtt{y} \geq 0)$
b	Button pressed	`button=true`
o	Postcondition holds	$(\mathtt{pt.x}{=}\mathtt{x}) \wedge (\mathtt{pt.y}{=}\mathtt{y})$

At the root, the program is at p_0 with a valid input. The environment can choose to either enter valid input again, enter invalid input, or press the button. If the button is pressed, the system *attempts* to advance the program counter to p_1; however, the aspect is able to interrupts it, moving to a_0. Since the input is still valid, the aspect executes `continue`, which returns control to the system at p_1, allowing it to execute `movePoint`; since the aspect has no power to interrupt once the call has actually been reached, the system is able to reach p_2 with the postcondition satisfied.

However, in the middle subtree, we see what happens when the environment invalidates the input, and then presses the button; the aspect takes control, but since the input is not valid it executes `skipover`, returning control at p_2 without allowing the program to execute `movePoint`. From the system's perspective, the aspect could have chosen to allow this transition: this possibility is indicated by a dotted transition. The aspect thus has the capability of preventing $p_1 \wedge b$ being true while r is false.

This translation is simplified, and does not take into account the possibility of other aspects taking control both before and after the guarantee aspect. Since the ATSs are compiled from code, the control-flow of the code is represented using a program counter variable; however, if ATSs are used in high-level modelling (aspect-oriented *design*), then this can be dispensed with. Formal support for such design is another goal of this work. It is not necessary for the model to be finite-state, as it is here, but finiteness is a sufficient condition for model-checking to be decidable.

4. REASONING WITH ASPECTS

In this section we discuss formal specification in alternating-time logic, and compositional reasoning about ATS translations of aspect-oriented programs.

4.1 Specification

The aspect we describe is introduced in order to guarantee a simple temporal safety property: it is *never* possible to reach a state which is a call to `movePoint` (p_1 in Figure 1) while the precondition is violated. This is a property which should hold in all executions of the system, regardless of nondeterministic choices.

A classical state machine model allows us to quantify over possible executions using temporal logic [4]. Is there an execution where the point is never moved? Certainly, the user need never press the button. Does the point pass through (2,3) in every execution? No: there are sequences of inputs which prevent this. An ATS model encodes not only this information, but also *which agents* co-operate to create an execution. In this case, both of the executions we have discussed are enforced by the environment alone: it is able to prevent the point from ever moving, while the system must move the point whenever the button is pressed.

We say that the environment has a *strategy* – a way to resolve the choices available to it – which prevents the point from moving. At any given time step in the program, the environment never has control over the entire program state: only the two fields and the button. How the rest of the state evolves is up to the system. However, with the system specified as it is, this limited control suffices for the environment to keep the point still. It is often desirable to show that environment properties are preserved [3].

Definition 4.1 (Strategies and Paths). *Given an ATS (Ω, S, P, L, R), a* strategy *for agent $a \in \Omega$ is a map $f_a : S^+ \to 2^S$, such that for all $w \in S^*$ and $s \in S$, $f_a(ws) \in R(s, a)$.*

The choice of a strategy by an agent constrains the possible executions. Given a strategy f, $\pi_f \in 2^{S^\omega}$ is the set of all infinite paths which the remaining agents are able to choose.

Alternating-time logic is defined with respect to strategies and the paths they determine. For example, given a formula φ:

- $s \models \langle a \rangle \mathbf{X}\varphi$ if a has a strategy f such that for all paths $st\ldots$ in π_f, $t \models \varphi$
- $s \models \langle a \rangle \mathbf{F}\varphi$ if a has a strategy f such that for all paths $st_0t_1\ldots$ in π_f, there is an i such that $t_i \models \varphi$

The purpose of adding the precondition-guarantee aspect is to impose a new property which did not hold before: that the environment *cannot* cause `movePoint` to be called without the precondition holding. It may still press the button, but the aspect alters the system's response, skipping over the call to `movePoint` if the precondition is violated[1]. At the same time, the aspect should not prevent `movePoint` from being called if the precondition does hold. An aspect's specification, then, as has been observed [12], is twofold: it has new properties that it must guarantee, and old properties that it must preserve. Both are dependent upon being used according to a contract.

We are transforming the system's code to meet the specification. To encode at the state-machine level the fact that we are doing this with an *aspect* rather than a *code patch* – that is, for modularity and maintanability rather than correctness – we represent this modification as the actions of another agent (the aspect) which is able to seize control of the system's program-counter at certain points, execute some code, and then return control.

4.2 Compositional Reasoning

How, then, do we show that the aspect meets both parts of its specification? The simplest approach to verifying a program is to create a formal model of the entire program, and prove it meets its specification. In general this is not feasible for two reasons: first, a state-machine translation grows exponentially in size with the number of variables, and rapidly becomes far too large for available memory; second, the reasoning task becomes forbiddingly complicated and cannot be decomposed or distributed in any way.

So it has been held that reasoning should be modular: and the most straightforward way to make reasoning modular is to follow the modular structure of the program. That is, given modules X and Y, which interact in a given way, we prove that if X behaves according to specification, so will Y; and if Y behaves according to specification, so will X. This is *compositional*, *modular* or *assume-guarantee* [10, 7] reasoning.

Since aspects form an alternative decomposition, it seems desirable to do reasoning that follows the aspect structure of a program. Not only would this facilitate reasoning about aspect-oriented programs, it would also promote more flexibile compositional reasoning in general, providing alternative decompositions which might be more amenable to proof. Our goal,then, is to develop compositional proof rules for aspect-oriented programs. The remainder of this section presents two such proposed rules.

4.2.1 Imposition Rule

Let M be the module, and F (for 'feature') be the aspect; they are attached using a binding c which identifies join-points in M with F's pointcuts (in our example, the calls to `movePoint` make up the pointcut `MP`). Consider first one half of the compositional reasoning task: assuming M is correct, and c is the correct binding, we wish to show that F must

[1]If the `Point` class were not a black box, an aspect could instead insert precondition-checking code at the beginning of `movePoint`, changing the class specification.

```
p'0:  goto p'0 □ goto p'1
p'1:  beforeJP
p'2:  pt.movePoint(x,y)
p'3:  afterJP
p'4:  goto p'0
```

Figure 3: Abstraction A_M of the program M: $\square$ indicates a nondeterministic choice

have the desired effect. That is, we wish to analyze F in isolation. However, since it is a controller which reacts to its environment – the program-counter changes of a module – it is, by itself, an open system, and needs to be closed with some model of its environment.

This model should be considerably simpler than M itself, since most of the behavior of M is of no interest to the aspect – only the entering and leaving of join-points. Such an abstraction A_M is shown in Figure 3; it is not much simpler than the program, but `button` has been abstracted away as irrelevant to the aspect's correctness. All that matters is that the abstraction calls `movePoint` some number of times with varying values of x and y.

Given this abstraction, we construct its composition $A_C(A_M, F)$; note that the composition C must be abstracted as well – since the pointcuts remain the same, but the join-points in a pointcut are different in M and A_M. The important property of this composition is that it preserves all the capabilities of F: the system is abstracted, but F's behaviors are neither expanded nor constrained. More formally, we say that $A_C(A_M, F)$ is an *S-abstraction* of $C(M, F)F$, which we write:

$$A_C(A_M, F) \leq_S C(M, F)$$

and that the aspect has the same capabilities (is A-equivalent):

$$A_C(A_M, F) \leq_S C(M, AF)$$

We refer to this composition as the *abstract aspect*, to follow the terminology of AspectJ; but note that it is not the aspect which was been abstracted!

Recall that the new property we wish to demonstrate is that the environment can be prevented from calling `movePoint` when the precondition is unsatisfied; that is, that the system and aspect together can prevent the call whatever the environment does. If we can demonstrate this for the abstract aspect, and show that it the abstract aspect has all of the system and aspect capabilities of the full composition, we can conclude that the full composition meets its specification.

Thus we state our first compositional reasoning rule, which we'll call the Imposition Rule, used to prove that an aspect guarantees a capability $\langle S, A\rangle\varphi$ of S and A:

$$\frac{A_C(A_M, F) \models \langle S, A\rangle\varphi \qquad A_C(A_M, F) \leq_{\{S,A\}} C(M, F)}{C(M, F) \models \langle S, A\rangle\varphi}$$

That is, if the abstract aspect meets the specification *and* it is an S, A-abstraction of the full composition with respect to S, then the full composition also meets the specification. By construction, the abstract aspect is A-equivalent to the full composition, and an S-abstraction, so the remaining obligations are to show that it is an S, A-abstraction (which is not necessarily implied by being both an S and A abstraction [2]), and that $\langle S, A\rangle\varphi$ holds in the abstract aspect.

To sum up, the Imposition Rule enables us to verify an abstract aspect, and show that the verification holds in the full composition by proving the abstraction relation between the abstract aspect and the full composition.

4.2.2 Preservation Rule

Of course, we also wish to show that the aspect preserves some of the existing properties of the base program. We take a similar approach: composing the module with an abstraction of the aspect, checking the desired property on this composition (which we dub the *program-in-context*), and using the abstraction relation to conclude that the property holds in the full composition, without actually needing to construct and analyze it.

The intuition behind the program-in-context is that it is the program in a very general aspect-oriented runtime environment, with the join-points fixed. At any join-point, the aspect may choose to interrupt, take control, and return to any allowable program point, possibly with some changes to program state. The program-in-context is *not* equivalent to the unmodified program: some capabilities of the system are likely to be broken by the imposition of the aspect.

The program-in-context $C(M, A_F)$ for our example is a composition of the base program with the following abstraction:

```
over mp(x,y) {
  continue [] skipover;
}
```

In other words, it is an A-abstraction of the full composition. Note that this embodies two additional assumptions: the aspect does not modify any state readable to the program, and that it only inserts before and over advice – never after. These assumptions are necessary for the program-in-context to preserve the desired property.

Part of the ATS for the program-in-context appears in Figure 4. In the unmodified system, whenever the PC is at line p_1 with the precondition satisfied, the postcondition is satisfied at the next step. This system capability is stated in ATL as:

$$\langle S\rangle p_1 \wedge r \Rightarrow \mathbf{X}o$$

We must check the program-in-context to see that this system capability is also preserved there. Looking at 4, we see that although the aspect can prevent the system getting to p_1, once it is there it can execute `movePoint` and satisfy the precondition.

So we state the Preservation Rule: if system capability $\langle S\rangle\varphi$ holds in the program-in-context $M \parallel_c A_F$, and the program-in-context is an S-abstraction of the full composition, then we can conclude that $\langle S\rangle\varphi$ holds in the full composition:

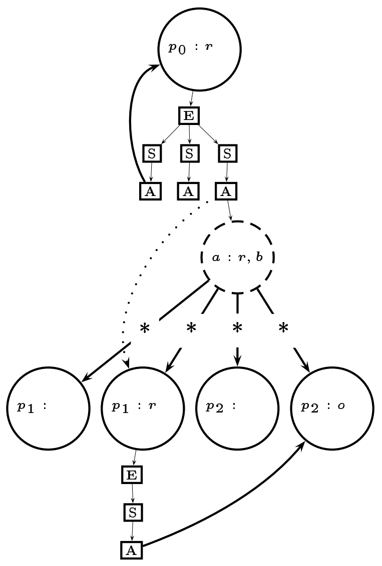

Figure 4: Fragment of ATS for program-in-context

$$\frac{C(M, A_F) \models \langle S\rangle\varphi \qquad C(M, A_F) \leq_S C(M, F)}{C(M, F) \models \langle S\rangle\varphi}$$

5. RELATED WORK

Researchers studying the *feature interaction problem* have encountered similar issues, since features often have similarly cross-cutting effects. Ryan et al. [11] have developed a *feature construct* for state-based modeling languages; their features are expressed at the state-machine level of abstraction rather than the program-code level. They have also [3] used an alternating transition system framework to prove that imposition of a feature maintains desirable properties of a system, even though it is a non-monotonic composition in general. The difference is that in their formalization, each module is represented by an agent; this allows for reasoning about capabilities of agents before and after feature imposition, but does not allow for the distinction of base and feature as separate agents, and thus does not lead to the kind of modular reasoning we wish to do.

Clifton and Leavens [5] address modular reasoning with aspects, and suggest two types of explicit contracts: observers and assistants. An observer is an aspect which does not change the existing specification of any module it is attached to; it only ever changes its own state. That is to say, the capabilities of the module by itself are identical to the capabilities of the composition of module and observer; only the aspect gains new capabilities. Assistants may modify system capability.

Fisler and Krishnamurti et al. [6] also use a state-based model of feature composition, and aim towards compositional reasoning. They have an effective decision procedure for proving that a feature preserves and guarantees properties without needing to construct the full state space; however, they do not consider the situation where a feature can disable a transition of the system it modifies.

We are not aware of any work which formalizes, in a general way, the weaving of aspects in general, dealing with cases like weaving of class hierarchies or data-flow graphs that are not handled by the proposed method.

6. CONCLUSION AND FUTURE WORK

We have discussed a proposed approach to modular reasoning with aspects. This approach is a variety of assume-guarantee reasoning, using an alternating transition system model, with alternating temporal logic [1] as a specification language. We have illustrated the fundamentals of this approach on an example, demonstrating how the proposed formalism enables compositional reasoning.

This work is in its early stages, and there is much to be done. Proving the necessary abstraction relations is the difficult part of the approach, and this must be shown to be scalable and relatively automatable. Further, the decidability of the analysis depends on a finite-state model, and so abstractions of the state-space are necessary to make a general program finite-state; any interactions between these abstractions and those of the proposed compositional rules must also be considered.

7. ACKNOWLEDGMENTS

I thank Marsha Chechik for extensive assistance with the ideas and presentation and Steve Easterbrook for fruitful discussions. Shmuel Katz started me thinking about the problem, and directed me to the literature on superpositions. The reviewers' comments helped to improve and clarify the material.

8. REFERENCES

[1] Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. "Alternating-time Temporal Logic". In *Proc. 38th IEEE Symposium on Foundations of Computer Science*, pages pp.100–109, 1997.

[2] Rajeev Alur, Thomas A. Henzinger, Orna Kupferman, and Moshe Y. Vardi. "Alternating Refinement Relations". In *Proceedings of CONCUR '98*, pages 163–178, 1998.

[3] F. Cassez, M. D. Ryan, and P.-Y. Schobbens. "Proving Feature Non-interaction with Alternating-Time Temporal Logic". In S. Gilmore and M.D. Ryan, editors, *Language Constructs for Describing Features.* Springer-Verlag, 2001.

[4] E. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 1999.

[5] C. Clifton and G.T. Leavens. "Observers and Assistants: A Proposal for Modular Aspect-Oriented Reasoning". In *Proceedings of FOAL 2002 (Foundations of Aspect-oriented Languages)*, 2002.

[6] K. Fisler and S. Krishnamurthi. "Modular Verification of Collaboration-Based Software Designs". In *Proceedings of FSE '01*, September 2001.

[7] Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. "You Assume, We Guarantee: Methodology and Case Studies". In *Proceedings of CAV '98*, pages 440–451, 1998.

[8] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. "An Overview of AspectJ". *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[9] John Mitchell. *Foundations for Programming Languages.* MIT Press, 1996.

[10] Corina S. Pasareanu, Matthew B. Dwyer, and Michael Huth. "Assume-Guarantee Model Checking of Software: A Comparative Case Study". In *Proceedings of the 1999 SPIN Workshop on Software Model Checking*, 1999.

[11] M. Plath and M. Ryan. "Feature Integration using a Feature Construct". *Science of Computer Programming*, 41(1):53–84, 2001.

[12] M. Sihman and S.Katz. "A Calculus of Superimpositions for Distributed Systems". In *Procedings of AOSD 2002*, 2002.