Concern specific languages and their implementation with abc

Eric Bodden
Chair for Computer Science II
Programming Languages and Program Analysis
RWTH Aachen University
52062 Aachen, Germany
ericbodden@acm.org

ABSTRACT

In this work first we introduce the notion of concern specific languages (CSL) which are to a specific crosscutting concern, what domain specific languages are to a specific domain. Implementing such CSLs was a tedious task in the past since no extensible frameworks for implementing crosscutting concerns existed. With the AspectBench Compiler (abc) [1], which was released in October, researchers now have a powerful extensible compiler for the aspect-oriented language AspectJ [6], enabling easy implementation of language extensions or even whole CSL for a specific crosscutting concern. We first motivate CSLs in general and give examples of such languages which exist already. In the subsequent chapters we introduce one specific CSL and report on our implementation using abc and specifically about how CSLs can interact with and reuse each other. We will see that the use of CSLs in general provides better comprehensibility and analyzability.

1. CONCERN SPECIFIC LANGUAGES

Through the advances in model and domain driven development in combination with more extensible compiler frameworks and code generation technologies, we have seen a rise of domain specific languages (DSL) for the most different applications during the last decade. Once given an implementation of a DSL, such a language is easier to understand and maintain with respect to the problem it solves [12]. DSLs have the advantage that they use a notation which comes close to notations that people who are concerned with the problem domain are familiar with. This enables a straightforward translation from specification and models into the DSL followed by an automatic conversion into a running implementation.

Van Deursen et al. [12] refer to a domain specific language as a "programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain".

Similarly we define a concern specific language as follows:

A concern specific language is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to or in support of, a particular crosscutting concern, comprising implicit or explicit quantification over events in the dynamic control flow.

Concern specific language extensions are likewise defined.

This reflects another tendency in software engineering, which came up during the last years: the wish to separate *crosscutting concerns*. Such concerns typically consist of code that functionally represents a single unit but, due to limitations of the implementation language, is usually scattered throughout the whole application. Aspect-oriented programming (AOP) tries to overcome this problem by separating such concerns into single units called *aspects* and has proven very powerful for the injection of mostly non-functional properties into applications. Essential to AOP is the ability to quantify [4] over events in the control flow of an application. This enables on to apply functionality not only at certain singular events but rather whole sets of such.

Thus, in AspectJ, one is able to build expressions similar to regular expressions, which reason about the static structure (e.g. every point within a certain class) as well as the execution flow (e.g. all calls to a certain method within the control flow of another). This is usually powerful enough to refactor out most crosscutting concerns from a given business application.

However there are often situations where the AOP languages, which are available today, simply do not match very well the concern one seeks to implement. The concern is actually modeled in high level of abstraction but when it comes to the implementation of that concern, one has to model it again, this time in code of some AOP language, which often results in dozens of lines of code for a single entity in the original model. CSLs provide an additional, valuable layer of abstraction. In the following we give some examples of past and currently ongoing research that tries to overcome this problem by extending AOP languages or building even new languages. They make use of AOP as an implementation strategy in order to reuse the power of AOP languages with respect to quantification.

1.1 Stateful aspects

SPLAT Mar. 15th, 2005, Chicago, USA.
ACM 0-89791-88-6/97/05.

The AOP implementation JAsCO [11], for instance, enables a simple notation of stateful aspects [3]. Such a notation is not domain specific. Stateful aspects are useful whenever state of the system behavior need to be tracked. This could aid implementation of concerns as e.g. verification, as described later. The example models a finite state machine

Table 1: Stateful aspects in JAsCO

```
start>p1;
p1: execute(startmethod) > p3||p2;
p3: execute(stopmethod) > p1;
p2: execute(runningmethod) > p3||p2;

after(): p2() {
  //do something
}
```

starting in state `p1`. When in `p1`, it is tried to match the pointcut `execute(startmethod)`. When that happens, the state is changed nondeterministically to `p3` or `p2`. Transition at `p3` and `p2` are enabled; it is tried to match either the pointcut `execute(stopmethod)` or `execute(runningmethod)`, and so forth. Each state change can be linked to a piece of advice as shown above with `after(): p2()`.

The language is a CSL because its purpose is the support of stateful aspects. The typical quantification is built in through the use of pointcuts. With respect to verification, this approach could be used to easily model finite state machines which in turn model logical formulas.

1.2 Association aspects

Sakurai et al. [9] propose association aspects which can be used to easily implement and establish relations between arbitrary objects in an application. The example here (Table 2) implements an equality relation. It can be instantiated using

```
Bit b1 = new Bit(), b2 = new Bit();
Equality a1 = new Equality(b1,b2);
```

This calls the explicit constructor of the following aspect.

Table 2: Association aspects by Sakurai et al.

```
aspect Equality perobjects(Bit, Bit) {
  Bit left, right;
  Equality(Bit l, Bit r) {
    associate(l, r); //establishes
    left = l; right = r; //association
  }
  after(Bit l) : call(void Bit.set())
    && target(l) && associated(l,*){
    propagateSet(right); //when left is called,
  }                      //call set on right
  after(Bit r) : call(void Bit.set())
  && target(r) && associated(*,r){
    propagateSet(left); //when right is called,
  }                     //call set on left
  //helper methods go here
}
```

In the constructor, `associate(l,r)` is invoked, which associates the new aspect instance with the input objects `l` and `r`. In pieces of advice, those references can be queried again: The first piece of advice executes whenever `Bit.set()` is invoked on the instance that is associated as `l`. In this case, the value of `l` is propagated to the right Bit. The second piece of advice behaves the other way around.

Association aspects are a CSL because their purpose is to allow the implementation of crosscutting concerns which are related to specific objects at runtime. Association aspects allow not only quantification over events but also over sets of objects, assuming that for instance the above relation could be instantiated for entire sets of objects by certain pieces of advice. With respect to verification, this approach could be used to associate a stateful aspect (e.g. a finite state machine as above) with a certain set of objects.

1.3 Rich pointcut models

Ostermann and Mezini [7] propose for their programming language ALPHA pointcuts holding expressions in the Prolog logical programming language and are even more powerful (Turing complete). They enable reasoning about certain properties of the static and dynamic structure of a program (table 3). Through the use of Prolog however, they provide a language that is extensible by itself by adding new rules, which is not the case for the other approaches we found.

Table 3: *reachable* pointcut definition in ALPHA

```
reachable (Obj1,Obj2) :- store (Obj1, ,Obj2).
reachable (Obj1,Obj2) :- store (Obj1, ,Obj3),
  reachable (Obj3,Obj2).
```

Here an object is `reachable` from another, if it is in the transitive hull of the `store` relation. (`store(Obj1, ,Obj2)` means that `Obj2` is a field value of `Obj1`.)

So in general one can conclude that there seems to be a clear interest in more expressive crosscutting languages and as a consequence in concern specific languages as well. This is due to the fact that CSLs offer a high level of abstraction, which enables with information hiding and high expressiveness with respect to a particular concern.

As an example, in this paper, we will consider the concern of Runtime Verification (RV), which is in general applicable to any problem domain but always implements the same functionality: to check if a program satisfies certain formalized conditions at runtime. This concern is usually not only scattered throughout the whole application (RV could be applied to any part of the system) but also would its support aid any possible software development process. Thus a language that eased implementation of this concern would not be domain specific but rather specific to this single concern.

To our best knowledge there are no Java-based RV approaches around at the moment, that would provide the same semantic power, our system provides to the user. This is because through the use of AOP we are able to use quantification in an implicit, elegant way, by the means of pointcuts. Indeed RV is a truly crosscutting concern with a strong need for such quantification since it affects the whole system. Stolz and Huch [10] showed for the functional programming language Haskell, that there the language itself is extensible enough to provide powerful verification at runtime, because means to implement quantification are already built in.

In non-functional languages as Java, however, without support for such crosscuts, implementation of RV is a tedious task, if not infeasible at all.

AOP languages as AspectJ give a first necessary level of abstraction. However as we will show, CSLs add another valuable such level, which provides for better understandability and maintainability. Thus we present an implementation based on a combined approach of a custom CSL with AspectJ backend.

2. AN EXAMPLE LANGUAGE: LTL OVER POINTCUTS

As concern-specific language, we define a linear-time temporal logic (LTL, [8]), which is usually the language of choice for reasoning about paths (here specifically about runtime execution paths), over pointcuts (see also [2]). We do so by adding the operators of the next-free[1] linear-time temporal logic to the usual pointcut designators.

2.1 Syntax and semantics

The temporal operators of LTL are F, G and U. Those can be cascaded and take usual AspectJ pointcuts as propositions.

For any given formulas or propositions φ and ψ, we define the semantics of G, F, and U at a given joinpoint t as follows:

- $G(\varphi)$ is true iff φ holds *Globally* on the execution path.

  Example: Always, either the user is logged in or we see no call to `debit`.
  ```
  G( if(User.isLoggedIn())
   || !call(* Account.debit()) )
  ```

- $F(\varphi)$ is true iff φ holds *Finally* somewhere on the execution path.

  Example: At some point in time all locks are released (by execution the appropriate method).
  ```
  F( execution(* Locking.releaseAllLocks()) )
  ```

- $(\varphi U \psi)$ holds iff φ holds *Until* finally ψ holds somewhere on the execution path. It can be evaluated as: $(\varphi U \psi)_t = \psi \vee (\varphi \wedge (\varphi U \psi)_{t'})$ for t' being the next joinpoint after t in the current control flow.

  Example: We do not see a call to `login` until at least 60 seconds after the last login attempt.
  ```
  (!call(* User.login(Credentials))) U
  (if(Time.getTime() > User.lastLogin()+60000))
  ```

In addition, we allow composition of those constructs (table 5) as well as free variables in such formulas, which are internally bound by the contained pointcuts. In the example shown in table 4, `o1`, `o2` and `bool` are such variables.

3. IMPLEMENTING THE LANGUAGE EXTENSION

In general we implement the verification of a given formula by creating the appropriate Büchi automaton (see figure 1) and modeling it in a given base language. This could be

[1]We exclude the usual neXt operator because reasoning about the *next joinpoint* does lead to a lot semantic difficulties for the user.

Table 4: Example using variables: (meaning *Object.equals(Object) is reflexive)*

```
Object o1,o2; boolean bool:
  G (
      (call(boolean Object.equals(Object))
          returning bool
        && target(o1) && args(o2)) ->
      (o1==o2 -> bool == true)
    )
```

Table 5: Example composing temporal operators (meaning *all acquired locks are finally being released*)

```
Lock l:
  G ( !call(* Lock.acquire(l)) ||
      F ( call(* Lock.release(l)) ) )
```

AspectJ but also another CSL. Note that in this specific example, one instance of the automaton has to be *associated* with each appropriate `Lock` object.

Given the examples of concern specific languages above, one can easily estimate that it would be very sensible to have all those languages or language extensions under one common framework. If that was the case, the implementation would had been much more straightforward. In the following we first present an implementation using "low level" AspectJ and in the subsequent chapter we demonstrate, how the use of other CSLs could improve comprehensibility and maintainability.

For our implementation of the runtime verification engine described above, we decided to use the open AspectJ compiler framework abc, which builds a very good platform for implementing concern specific languages and language extensions, since those can simply inherit implementation of crosscutting functionality from the AspectJ base implementation.

3.1 Implementation with abc using plain AspectJ

In general, for implementing a CSL with abc, the following steps are necessary:

- define keywords (lexer)
- define syntax (grammar)
- define semantics (AST rewriting passes or generation of AspectInfo)

When defining the actual semantics, the implementation strategy depends on the nature of the CSL. If the language defines semantical differences to the AspectJ base implementation, the appropriate *AspectInfos* have to be modified. An AspectInfo implements the semantics of an AspectJ constructs in pure Java. Otherwise, if the AspectJ base semantics are untouched, simple AST rewriting passes suffice, which in the end lead to a valid AST for the AspectJ programming language, which can then be further processed automatically.

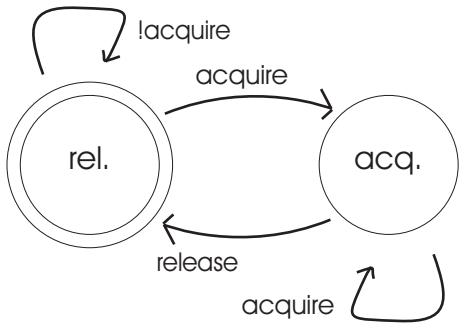

Figure 1: Simple automaton for formula in table 5

For our specific case, the second approach suffices. The rewritten AST corresponds to the following AspectJ code for the example formula.

```
aspect Formula issingleton {

  List<Lock> affectedObjects$1;

  int Lock.state$Formula = 1;

  pointcut p1(Lock l): call(* Lock.acquire(l));
  pointcut p2(Lock l): call(* Lock.release(l));

  after(Lock l): p1(l) {
    switch(l.state$Formula) {
      case 1: l.state$Formula = 2;
      case 2: l.state$Formula = 2;
    }
  }

  after(Lock l): p2(l) {
    switch(l.state$Formula) {
      case 1: l.state$Formula = 1;
      case 2: l.state$Formula = 1;
    }
  }

  //associate aspect
  before(Lock l): p1(l) || p2(l) {
    if(!affectedObjects$1.contains(l)) {
      affectedObjects$1.add(l);
    }
  }

  //assume pointcut "shutdown" given
  before(): shutdown() {
    for(Lock l: affectedObjects$1)
    {
      if(l.state$Formula==2)
        Verifier.reportError(this);
    }
  }
}
```

As one can easily see, this implementation naturally suffers from several drawbacks compared to the one presented first: On the one hand the state has to be tracked manually. Here we decided for a state variable introduced by the aspect on the object whose state is actually being affected by the evaluation. Another implementation strategy could employ lists or hash tables for this purpose but that would even be more tedious.

On the other hand we had to associate the aspect manually with the objects it should track by putting it into a list. With respect to efficiency this may make no difference, since association aspects internally rely on lists and hash tables, too, however, the mental effort an implementor needs to invest for this specific implementation seems to be much larger than using the approach building on top of existing CSLs.

3.2 Possible implementation using stateful aspects and association aspects

Employing both, association aspects an stateful aspects, would ease development in the following way: Association aspects allow one to associate a formula, that is to be checked, directly with the objects which it deals with. Employing stateful aspects, the appropriate automaton can be modeled in a concise and readable form.

```
aspect VerificationAspect {
hook Formula {

Formula(Lock l) {
  associate(l);
  //aspect states
  start>rel;
  rel: call(* Lock.acquire(l)) > acq;
  rel: !call(* Lock.acquire(l)) > rel;
  acq: call(* Lock.release(l)) > rel;
  acq: call(* Lock.acquire(l)) > acq;
}

//assume pointcut "shutdown" given
before(): shutdown() {
  if(!inState(rel))
    Verifier.reportError(this);
}

//automatic instantiation
before(Lock l): call(* Lock.acquire(l)) {
  new Formula(l);
}

}}
```

Unfortunately, JAsCO build an own implementation of AOP and association aspects were implemented as a prototype to the original AspectJ implementation ajc [5].

In the following we would like to perform a short evaluation of the benefits of the latter approach:

4. RESULTS

4.1 Comprehensibility/Maintainability

We see a largely increased comprehensibility by the use and combination of the proposed CSLs. The version employing both, stateful aspects and association aspects, proves to be much more intuitive than the raw ApsectJ approach. Codebloat is reduced to a minimum.

4.2 Evolvability

The combination of different CSLs can facilitate, as we showed, the development of new CSLs a lot. Thus we conclude that a repository of such CSLs can lead to a system that is very evolvable as a whole. Approaches as described in [7] go still a step further by allowing to extend the language by its own language constructs.

4.3 Modularity

We conclude that modularity is neither increased nor decreased by the use of CSLs with respect to a low level AOP approach, since each aspect in a CSL can usually be mapped uniquely to one aspect in the AOP base language.

5. CONCLUSION

In this work we introduced a definition of concern specific languages. We showed through several examples that a lot of research with respect to such languages is already underway. Using the concern of runtime verification as an example, we proposed a powerful CSL to model *and implement* the problem. The proposed LTL is a premier example for such a CSL since its power derives from the ability to quantify over whole sets of events. We gave an outline of possible implementation strategies using the abc compiler framework, noting that researchers and practitioners could benefit a lot from reuse of CSLs since they may well work together or be implemented on top of each other. This strategy enhances comprehensibility, while the use of AOP as such increases modularity. abc is generally capable of and suitable for such approaches, due to its own high degree of modularity and built in support for crosscutting concerns.

6. ACKNOWLEDGEMENTS

I wish to thank Volker Stolz and Friedrich Steimann for reviewing an initial draft of this work. Also I wish to express my gratitude to the whole abc team for providing their framework and for helping me to extend it.

7. REFERENCES

[1] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhot, O. Lhot, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. An extensible AspectJ compiler. In *Proceedings of the Fourth ACM SIG International Conference on Aspect-Oriented Software Development (AOSD'05)*, Chicago, USA, March 2005. ACM Press.

[2] E. Bodden. A lightweight LTL runtime verification tool for Java. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 306–307. ACM Press, 2004.

[3] R. Douence, P. Fradet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 141–150. ACM Press, 2004.

[4] R. Filman and D. Friedman. Aspect-oriented programming is quantification and obliviousness, 2000.

[5] E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In G. C. Murphy and K. J. Lieberherr, editors, *AOSD*, pages 26–35. ACM, 2004.

[6] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *ECOOP*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer, 2001.

[7] K. Ostermann and M. Mezini. Design and implementation of pointcuts over rich program models. *TechReport TU Darmstadt*, 2005. http://www.st.informatik.tu-darmstadt.de/.

[8] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science*, pages 46–57. IEEE Computer Society Press, 1977.

[9] K. Sakurai, H. Masuhara, N. Ubayashi, S. Matsuura, and S. Komiya. Association aspects. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 16–25. ACM Press, 2004.

[10] V. Stolz and F. Huch. Runtime verification of Concurrent Haskell programs. In *Proceedings of the Fourth Workshop on Runtime Verification*, volume 113 of *ENTCS*. Elsevier Science Publishers, 2004.

[11] D. Suvée, W. Vanderperren, and V. Jonckers. JAsCo: an aspect-oriented approach tailored for component based software development. In *AOSD*, pages 21–29, 2003.

[12] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, 2000.