Assessing Properties of Aspect-Based Components

Giovanni Denaro Mattia Monga

June 17, 2001

1. Introduction

Distributed systems have to cope with the issues related to concurrency, security, and network dissemination. Different properties of the system are relevant when different perspectives of a system are considered [3]. For example, as far as concurrency is concerned, the absence of *deadlocks* [10] is a relevant property of a system. Similarly, considering security issues, one can be interested in verifying that the *least privilege principle* [13] is guaranteed. In principle, this properties depend just on the code that implements the specific concern, i.e., concurrency and security. However, the relevant code is generally scattered across all the components of the system.

Aspect-oriented approaches aim at factoring out those concerns that cross-cut an entire software system, or parts of it [7]. Aspect-oriented languages allow to code solutions to scattered problems in somewhat untangled units– called *aspects*–, avoiding the scattering of relevant code across multiple components [11]. Aspects can be developed in isolation and then *woven* into the functional code to obtain a complete system.

In this paper we argue that the increasing availability of the aspect technologies [6, 9] makes it feasible to analyze specific properties focusing just on aspect modules. This paper proposes the idea that the analysis of aspect-oriented units can be the source for derivation of verification models. In our position, if an aspect is really encapsulating a non-functional concern of the system in a separated component, it should be possible to discover the relevant properties of the encapsulated concern just by examining such component. We suggest an approach to the general problem and we show how it applies in the case of proving deadlock freedom of a concurrency aspect coded with AspectJ [6].

The rest of the paper is organized as follows: Section 2 sketches the proposed verification approach. Section 3 illustrates the simple experiment that exploits such an approach. Finally, Section 4 draws some considerations and remarks.

2. The Approach

The approach proposed in this paper analyzes aspect-based software components

aiming at verifying safety properties related to specific concerns of software systems. In our perspective, aspects are independent components, which can be *woven* to different (and possibly third party developed) software systems to evolve their functionalities. The proposed approach aims at assessing general properties of aspects without considering the environments in which they can be used.

The aspect code is analyzed in order to derive models that are suitable for applying formal verification techniques. Properties related to the analyzed concerns have to be formally specified as well. A proper formal technique is then applied in order to either demonstrate that the specified properties hold or provide counterexamples of undesired behaviors. The steps of this approach, exemplified in the case of aspects for concurrency related concerns, are:

- select some general properties that are relevant for a given concern of software systems, e.g., deadlock freedom or liveness of states for concurrency;
- select a formal verification technique suitable for analysing the selected properties of interest, e.g., model checking;
- identify the aspect that encapsulate the relevant code as far as the selected properties are concerned, e.g. the aspect coding the synchronization policy;
- use the aspect code for deriving the verification model required for applying the selected formal technique, e.g., PROMELA specifications [4] in the case of model checking by means of SPIN [5];
- specify, if needed, the selected properties with the proper formalism, e.g., linear temporal logic [12] for SPIN, and run the verification tools, e.g., SPIN;
- report on success of the verification or analyze the provided counterexamples, e.g., invalid executions leading to deadlock states.

We experimented this approach in the case of concurrency concerns described by means of AspectJ [6]. The SPIN model checker has been used to demonstrate the deadlock freedom property for a sample aspect coding a synchronization policy. We derive the PROMELA model needed by SPIN by examining the aspect code with a systematic procedure. At the end of the analysis we found a counterexample that leads to a deadlock state, thus proving that the investigated property does not hold in general for the analyzed aspect, although this does not exclude that specific systems using this aspect may work correctly. We are well aware that the achieved results are preliminary, but we believe they are convincing of the feasibility of the proposed approach. Next Section 3 reports on the experiment.

3. Preliminary Experience

Listing 1 shows two software modules, actually two Java classes, which handle file and directory objects from a hypothetical file system. In the former class (Listing 1.a), the method `removeUseless()` is designed for removing files that are marked as useless. This method checks if a given file is indeed useless and, if this test succeeds, removes it from the containing directory. In a concurrent system, this

operation must execute avoiding race conditions among running threads. To this end some synchronization code is needed. In particular, the programmer needs to be sure that, when a given file is recognized as useless, it does not get updated before being actually removed and the correspondent directory object does not get changed (e.g., deleted) during the operation. A possible implementation of this policy uses two locks for both the involved directories and files, and encloses the critical sections in synchronized blocks. Similar mechanisms are required for the method `updateDir()` of the class `User2` (Listing 1.b).

Listing 1: Code of the example

```
public class User1 {
  public void removeUseless(File f){
    if(f.isUseless()){
      Directory d = f.getDir();
      d.removeFile(f);
      System.out.println("Removed: "
                         + f.getName());
    }
  }
}
                                    // (a)

public class User2 {
  public void updateDir(Directory d){
    Enumeration e = d.getFiles();
    while(e.hasMoreElements()){
      File f = (File)e.nextElement();
        f.update();
        System.out.println("Updated: " +
                           f.getDir().getName()
                           + f.getName());
      }
    }
}
                                    // (b)
```

In order to be synchronized, the system containing the two classes `User1` and `User2` can be evolved by applying the synchronization policy coded in the aspect shown in Listing 2. The used aspect-oriented language is AspectJ [6]. In AspectJ the `pointcut` keyword declares expressions that identify a set of points of the code (called *join points*). At these points arbitrary code can be injected. For each pointcut, an aspect also specifies the code to inject and the way of injection, e.g., new code can be injected `around` the code of the identified join point. In Listing 2, for instance, a pointcut called `file` (Listing 2, line 4) is declared that identifies the points where the method `removeUseless()` executes (Listing 2, line 7). The aspect code associated with the pointcut `file` locks the actual file object (Listing 2, line 16) and releases it when the wrapped functional operation (Listing 2, line 19) is completed. The pointcut `file` also identifies the executions of the method `update()` (Listing 2, line 6) that are synchronized as well. Analogous synchronization machinery is provided by the pointcut `dir` (Listing 2, line 24).

Listing 2: The concurrency aspect

```
1  aspect ConcurrencyAspect{
2
3   /* pointcut declarations */
4      pointcut file(File f):
5     (instanceof(f)
6      && executions(void File.update()))
7      || executions(void User1.removeUseless(f));
8   pointcut dir(Directory d):
9     (instanceof(d)
10     && executions(boolean
11                   Directory.removeFile(File)))
12     || executions(void User2.updateDir(d));
13
14  /* code to be woven when pointcuts occur */
15  static around(File f) returns void: file(f){
16       synchronized(f){
17         System.out.println("locked: "
18          + f.getName());
19         proceed(f);
20         System.out.println("unlocked: "
21          + f.getName());
22       }
23  }
24  static around(Directory d)
25    returns boolean: dir(d){
```

```
26        synchronized(d){
27          boolean b;
28          System.out.println("locked: "
29           + d.getName());
30          b = proceed(d);
31          System.out.println("unlocked: "
32           + d.getName());
33          return b;
34        }
35    }
36  }
```

A running system that instances objects from classes `User1` or `User2` merged with the aspect `ConcurrencyAspect` gets locks on the involved files and directories, potentially causing deadlock situations [8]. Therefore, a relevant property for this software system is *deadlock freedom,* and it should be possible to limit the scope of the analysis of this property to the aspect code. It is worth noting that the binding between the actual functional code and the code of the aspect is established by lines 5–7, 9–12 in Listing 2. Therefore, these statements have to be ignored in a general demonstration as the one we aim at providing.

A formal proof of deadlock freedom based on the aspect should address the synchronization policy independently from the actual functional code of the analyzed system. In other words, if the same aspect would be used to synchronize a different system, the previous demonstration should still apply.

3.1. The model

In order to prove the deadlock freedom, we modeled the aspect in Listing 2 in PROMELA [4] (for the sake of completeness the PROMELA specification is reported in Appendix A).

The procedure to derive the model can be summarized as follows:

- The existing resources, files and directories, are modeled with sets of boolean variables that are set to true when the corresponding object is locked and to false otherwise. (see Appendix A lines 6–7);
- The code related to each pointcut is modeled with a PROMELA process template. Actually, there are two process template associated with the pointcut `directory` and `file` (respectively, Appendix A lines 42–80 and lines 82–119). Each process locks a given resource (Appendix A lines 49–51 and lines 89–91) when it starts executing and release it (Appendix A line 78 and line 117) when it completes.
- The execution of the code associated to pointcuts is modeled by activating the corresponding process template.
- Since the code associated to different pointcuts is interwoven with functional code, it can be executed by simultaneous threads. The model makes the conservative hypothesis that between the locking and unlocking of a given resource, every other resource (including the one already locked) could be requested by other running threads. This is modeled by repeatedly instancing other processes chosen non-deterministically among all the available ones (Appendix A, lines 54–76 and lines 93–115). At each activation the resource to be acquired is chosen non-deterministically as well (Appendix A, lines 64, 74, 103, and 113).
- A bootstrap `init` process starts the acquisition of locks by non-deterministically activating the available process templates (Appendix A, lines 14–40).

Once every possible interaction among the pointcuts has been represented, the model checker handles non-deterministic choices by simulating all the possible state sequences. For this reason, the number of possible states needs to be bounded. In our model this is achieved by limiting the number of process activations (Appendix A, lines 3 and 4).

3.2. Deadlock freedom

The derived PROMELA model has been analyzed with the model checker SPIN and an invalid state has been discovered. This is enough to conclude that the synchronization policy coded by the aspect may lead to a deadlock. SPIN also computed the shortest path to the invalid state. In the example, the simplest deadlock situation occurs with the sequence of interactions [`ptcutFile` → `ptcutDir` → `ptcutFile`], when the file object on which the lock is acquired is the same in the first and the third pointcut.

4. Conclusions

This paper suggests that models suitable for verifying properties, which are relevant for specific concerns of a software system, can be derived from aspect-oriented units. This approach has the advantage that the system, but the aspect units, can evolve without affecting the validity of the verified properties. A preliminary assessment of this approach has been performed in the case of verification of deadlock freedom of a concurrency aspect. The proposed approach seems to be promising, although further work is needed. Currently, we are working on tuning the approach for making it applicable to more realistic case studies.

References

[1] T. S. Chow. Testing software design modeled by finite-state machines. *IEEE trans. on Software Engineering*, SE-4, 3:178–187, 1978.

[2] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *Proceedings of the 1999 International Conference on Software Engineering*, pages 285–294. IEEE Computer Society Press / ACM Press, 1999.

[3] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke. Viewpoints: A framework for integrating multiple perspectives in systems development. *International Journal of Software Engineering and Knowledge Engineering*, 1(2):31–58, 1992.

[4] G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.

[5] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997. Special Issue: Formal Methods in Software Practice.

[6] G. Kiczales, J. Hugunin, M. Kersten, J. Lamping, C. V. Lopes, and W. G. Griswold. Semantic-based crosscutting in aspectj^{TM}. In P. Tarr, W. Harrison, H. Ossher, A. Finkelstein, B. Nuseibeh, and D. Perry, editors, *ICSE 2000 Workshop on Multi-Dimensional Separation of Concerns in Software Engineering*, pages 69–74, Limerick, Ireland, June 2000.

[7] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Finland, June 1997. Springer-Verlag.

[8] D. Lea. *Concurrent Programming in Java*. Addison-Wesley Longman, 1997.

[9] K. J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, 1996.

[10] J. Magee and J. Kramer. *Concurrency: State Models & Java Programs*. Wiley, 1999.

[11] M. Monga. *Towards Software Components for Non Functional Aspects*. Phd program in computer and automation engineering, Politecnico di Milano, Italy, Jan. 2001.

[12] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS-77)*, pages 46–57, Providence, Rhode Island, Oct. 31–Nov. 2 1977. IEEE, IEEE Computer Society Press.

[13] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, Sept. 1975.

A. The PROMELA model

```
1  #define NDIRS 3
2  #define NFILES 3
3  #define MAXPROC 5
4  #define MAXSPAWN 1
5
6  bool dirLocks[NDIRS];
7  bool fileLocks[NFILES];
8
9  typedef pointcutParam {
10         byte lock;
11         chan sync;
12 };
13
14 init{
15  byte i = 0; byte j = 0;
16  byte spawned = 0;
17  chan wait1[MAXPROC] = [1] of { bit };
18  chan wait2[MAXPROC] = [1] of { bit };
19  pointcutParam r;
20
21  do
22  :: ( spawned < MAXPROC) ->
23     atomic {
24            r.lock = i;
25            r.sync = wait1[spawned];
26            run pointcutDir(r);
27            spawned = spawned + 1;
28     }
29  :: ( spawned < MAXPROC) -> i = (i + 1) % NDIRS
30  :: ( spawned < MAXPROC) ->
31     atomic {
32            r.lock = i;
33            r.sync = wait2[spawned];
34            run pointcutFile(r);
35            spawned = spawned + 1;
36     }
37  :: ( spawned < MAXPROC) -> j = (j + 1) % NFILES
38  :: break
39  od
40 }
41
42 proctype pointcutDir(pointcutParam p){
43  byte i = 0; byte j = 0;
44  byte spawned = 0;
45  show chan wait[MAXSPAWN] = [0] of { bit };
46  bool x;
47  pointcutParam r;
48
49  atomic {
50         (!dirLocks[p.lock]) -> dirLocks[p.lock] = true;
51  };
52
53
54  do
55  :: ( spawned < MAXSPAWN) ->
56     atomic {
57            r.lock = i;
58            r.sync = wait[spawned];
59            run pointcutDir(r);
60            spawned = spawned + 1;
```

```
61              wait[spawned−1]?x
62      }
63
64    :: ( spawned < MAXSPAWN) −> i = (i + 1) % NDIRS
65    :: ( spawned < MAXSPAWN) −>
66       atomic {
67              r.lock = j;
68              r.sync = wait[spawned];
69              run pointcutFile(r);
70              spawned = spawned + 1;
71              wait[spawned−1]?x
72      }
73
74    :: ( spawned < MAXSPAWN) −> j = (j + 1) % NFILES
75    :: break
76   od;
77
78   dirLocks[p.lock] = false;
79   p.sync!true
80  }
81
82  proctype pointcutFile(pointcutParam p){
83   byte i = 0; byte j = 0;
84   byte spawned = 0;
85   show chan wait[MAXSPAWN] = [0] of { bit };
86   bool x;
87   pointcutParam r;
88
89   atomic {
90          (!fileLocks[p.lock]) −> fileLocks[p.lock] = true;
91   };
92
93   do
94    :: ( spawned < MAXSPAWN) −>
95       atomic {
96              r.lock = i;
97              r.sync = wait[spawned];
98              run pointcutDir(r);
99              spawned = spawned + 1;
100             wait[spawned−1]?x
101     }
102
103   :: ( spawned < MAXSPAWN) −> i = (i + 1) % NDIRS
104   :: ( spawned < MAXSPAWN) −>
105      atomic {
106             r.lock = j;
107             r.sync = wait[spawned];
108             run pointcutFile(r);
109             spawned = spawned + 1;
110             wait[spawned−1]?x
111     }
112
113   :: ( spawned < MAXSPAWN) −> j = (j + 1) % NFILES
114   :: break
115  od;
116
117    fileLocks[p.lock] = false;
118    p.sync!true
119   }
```