LTS-based Semantics and Property Analysis of Distributed Aspects and Invasive Patterns*

Luis Daniel Benavides Navarro, Rémi Douence, Angel Núñez, Mario Südholt

OBASCO project; EMN-INRIA, LINA
Dépt. Informatique, École des Mines de Nantes
4 rue Alfred Kastler, 44307 Nantes cédex 3, France
{lbenavid,douence,anunez,sudholt}@emn.fr

Abstract. Invasive patterns are an extension of standard parallel and distributed architectural patterns for complex distributed algorithms. They have previously been implemented in terms of the AWED language, an aspect language with features for explicit distribution. In this paper we present two formal semantics based on labeled transition systems, one for AWED and the other for invasive patterns, for the definition of interaction properties of aspects and pattern compositions. We also show how the semantics can be used to check corresponding liveness and safety properties.

1 Introduction

In previous research we investigated how invasive patterns [1] that are defined on top of AOP languages can be used to address the complexity of distributed algorithms in real industrial middleware for replicated caching. We have implemented invasive patterns by means of a formal transformation into AWED [2], an aspect language with features for explicit distribution. The transformation defines invasive patterns in terms of interacting AWED aspects. This paper introduces two direct formal semantics, one for AWED and the other for invasive patterns. Furthermore, we harness these semantics to formally show, for the case of AWED, how remotely deployed aspects interact, and how individual pattern definitions interact in the case of invasive patterns.

During the design of invasive patterns we argued that distributed algorithms, and specific properties over those algorithms, can be analyzed and enforced easily in terms of invasive pattern abstractions. For example, Fig. 1 presents a high-level pattern-based view of the system structure of JBoss Cache's replication algorithm with transactions with pessimistic locking and a two phase commit protocol. A transaction is triggered by a specific method call represented by the first node in the pattern. Then successive calls to `get`, `remove` or `put` methods on the cache are executed and the information is stored for further replication. When a particular value is not present in the cache, the latter looks for the value

* Work partially supported by AOSD-Europe, the European Network of Excellence in AOSD (www.aosd-europe.net).

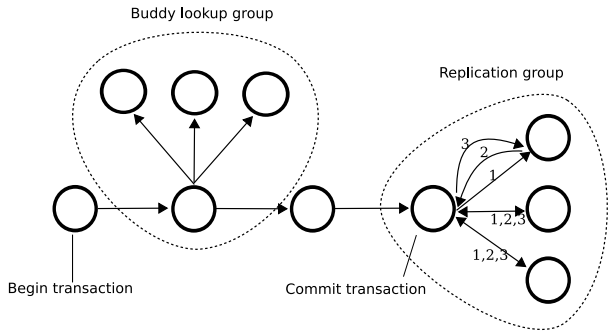

Fig. 1. Architecture of transaction handling with replication in JBoss Cache

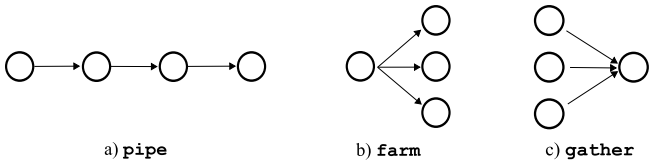

Fig. 2. Basic patterns

in a group of selected neighboring nodes, its so-called buddies, illustrated by the three edges starting in the second node of the figure. Once the end of a transaction is reached, the originating cache engages a two phase commit protocol. As part of this protocol, the originating cache sends a prepare message with the transaction control information (edges numbered 1), followed by answers from all buddies confirming agreement or non agreement (edges numbered 2). Finally, the originating cache sends a final commit or a rollback message depending on the answers it received (edges numbered 3).

As we have shown, the two phase commit protocol implementation can naturally be expressed in terms of compositions of invasive patterns, that is, extensions of the three basic patterns showed in Fig. 2 that account for highly crosscutting references of the patterns to contextual information. The steps denoted 1–3 in Fig. 1 correspond, for instance, to two applications of the `farm` pattern and one application of the `gather` pattern. Analyzing properties over such pattern specifications is more feasible than analyzing properties over current Java implementation of JBoss Cache (more than 50000, highly entangled, lines of code) where this architecture is essentially hidden. For that purpose, this paper presents three contributions: a formalization of AWED, a formalization of invasive patterns, both based on Labeled Transitions Systems (LTS), and some examples of how to use the LTS-based semantics with LTSA [3] (http://www.doc.ic.ac.uk/ltsa/eclipse) as analyzing tool.

2 The AWED Language

AWED supports AOP in a distributed context. A pointcut can match joinpoints on different hosts. A sequence of joinpoints can involve different hosts. An advice can be executed remotely, synchronously or asynchronously to the base execution. Furthermore, pointcuts can include predicates over groups of hosts.

The grammar shown in Fig. 3 shows the essentials of pointcut definitions in the AWED language (the full language definition can be found in [4]). The pointcut language allows matching of method calls (terminal `call`), nested calls (`cflow` means control-flow) and arbitrary (regular) sequences of method calls (non-terminal *Seq*). The constructors `host` and `on` specify (groups of) hosts where a pointcut is matched (or where an advice is executed). The constructors `target` and `args` bind values (such as the receiver or the arguments of a method call) to variables. This enables values to be passed from a matching execution event to the corresponding advice. Pointcuts can be composed using logical operators (union, intersection and complement). Sequences (*Seq*) are defined in terms of transitions of non-deterministic finite-state automata. An automaton is a set of transitions *Step*. Each transition has a label *id* and its pointcut *Pc* non-deterministically leads to a set of *Id*. The constructor `step` identifies the transition in the automaton that should trigger the advices.

// Pointcuts

Pc ::= `call`(*MSig*) | `cflow`(*Pc*) | *Seq*
| `host`(*Group*) | `on`({ *Hosts* })
| `target`({*Type*}) | `args`({*Arg*})
| *Pc* || *Pc* | *Pc* && *Pc* | !*Pc*

Seq ::= *Id*: `seq`({*Step*}) | `step`(*Id*,*Id*)

Step ::= *Id*: *Pc* → *Target*

Target ::= *Id* || ... || *Id*

Hosts ::= `localhost` | `jphost` | "*Ip*:*Port*" | *GroupId*

Fig. 3. The AWED language (excerpts)

3 Pattern Language

Figure 4 shows the syntax of invasive patterns language (we have omitted details for the sake of simplicity).

The pattern constructor `patternSeq` takes as argument a list $G_1\ A_1\ G_2\ A_2\ \ldots\ G_n$ of alternating group and aspect definitions. Each triple $G_i\ A_i\ G_{i+1}$ in this list corresponds to a pattern application that uses the aspect A_i to trigger the pattern in a source group G_i and realizes effects in the set of target hosts G_{i+1}. A group G is either defined as a set of host identifiers H or through a pattern constructor term itself. In the latter case, the group is defined as the source or target group of the constructor term depending on the argument position the term is used in. This constructor enables the definition of the basic patterns shown in Fig. 2. Pattern compositions can be defined with more complex `patternSeq` terms. For instance, the left hand side of Fig. 5 defines , using a

P ::= `patternSeq` G_1 A_1 G_2 A_2 ... G_n
G ::= H G | P G | ϵ
A ::= `aspect` { `around`((H, Id*)*): PCD $SourceAdvice$ [`sync`] $TargetAdvice$ }
PCD ::= `call`($MSig$) | `target`(Id) | `args`(Id+)
| PCD && PCD | PCD || PCD | !PCD
| Seq

Fig. 4. Pattern language

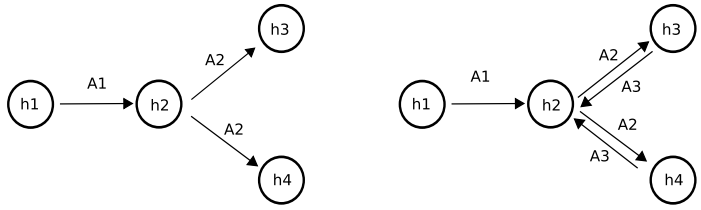

Fig. 5. Pattern Compositions

concrete syntax, a composition `pipe` then `farm`, and its right hand side defines a composition `pipe`, `farm` then `gather`.

Aspects A specify a pointcut PCD that allows the modularization of the crosscutting code that triggers a pattern, and define a source advice and a target advice executed respectively on the source and target groups of a pattern. Advice can be parametrized by source hosts H and bound values (see `args` below). A source advice can call the matched base call with the `proceed` keyword. Otherwise, the base call triggers the aspect but the execution of the corresponding base method is skipped. When a `sync` annotation is used to qualify target advice, the base program execution on source hosts is not resumed before the end of the target advice. The default behavior is asynchronous execution.

Pointcut definitions that, for presentation purposes, are essentially restricted to matching of method call joinpoints, may extract target objects with `target` and arguments of calls with `args` and use logical compositions of pointcuts. Following the paradigm of stateful pointcuts [5, 2] (and unlike AspectJ [6, 7]), pointcuts may match sequences (non-terminal Seq) of calls in the base program execution. We omit the syntax of sequences for presentation purposes.

4 Formal Semantics for Language Constructs

This section formally defines the semantics of our language constructs in terms of labeled transition systems (LTSs). It also shows how these definitions can be analyzed with model checkers such as LTSA. More precisely, we use Finite State Processes (FSP) [3], a textual description of LTSs. In the remainder we use the terms FSP and LTS interchangeably. We first focus on AWED in Sec. 4.1, then we extends its formalization for invasive patterns in Sec. 4.2.

```
1  // local aspect (and advice) to de deployed on h_1
2  Local = (h_1.loadInCache -> h_1.proceed ->
3           h_2.h_1.remoteAdvGo -> h_3.h_1.remoteAdvGo -> Local).
4
5  // remote advice to be deployed on h_2
6  RemoteAdv1 = (h_2.h_1.remoteAdvGo -> h_2.loadInCache -> RemoteAdv1).
7
8  // remote advice to be deployed on h_3
9  RemoteAdv2 = (h_3.h_1.remoteAdvGo -> h_3.loadInCache -> RemoteAdv2).
10
11 ||Aspect = (Local || RemoteAdv1 || RemoteAdv2).
```

Fig. 6. Semantics of a Cache Replication AWED Aspect

4.1 Formalization of (AWED) distributed aspects

AWED provides aspects in a distributed context by introducing two notions: group of hosts, and asynchronous remote advice. In AWED, an aspect specifies a group of hosts for pointcut and another group of hosts for remote advice. When a joinpoint occurs on *any* host of the pointcut group, the local advice is executed on this host, and the remote advice is executed on *each* host of the remote advice. Moreover, the remote advice is executed asynchronously (by default) and the local base program resumes its execution as soon as the remote advices are started. For instance, consider the following example:

```
aspect execution(void {h_1}.loadInCache(URL)) && args(url)
      { proceed(url); }
      { {h_2,h_3}.loadInCache(url)}
```

This distributed AWED aspect replicates the update on hosts `h_2` and `h_3` cache updates of host `h_1`. Indeed, when the pointcut corresponding to `loadInCache` is matched on `h_1`, the local advice `proceed(url)` performs the local cache update and the remote advice updates the caches on hosts `h_2` and `h_3`. Such an AWED aspect can be formally defined in LTS as in Fig. 6. An aspect is defined as the parallel composition of one LTS by host. An LTS is a finite state automaton with labeled transitions. For instance, in Fig. 6 there are three LTSs, the first one `Local` defines a sequence of four transitions each separated by `->`. The `Local` LTS models the aspect: when the joinpoint occurs on the first host (*i.e.*, `h_1.loadInCache`), the method is executed locally (*i.e.*, `h_1.proceed`), then the remote advices are started on the second and third hosts (*i.e.*, `h_2.h_1.remoteAdvGo` and `h_3.h_1.remoteAdvGo`). Then this recursive definition waits for the next occurrence of the pointcut. The definition `RemoteAdvice1` models the remote advice on the second host: it waits for `remoteAdvGo` to execute the replicated cache loading and waits again. The remote advice `RemoteAdvice2` on the third host is similar. The semantics of the full aspect is defined by the parallel composition (noted `||`) of these three LTSs. The resulting LTS `Aspect` is the synchronized product of the three automata and it implements rendez-vous for shared transitions.

The semantics of the woven program is defined by the parallel composition of the base program specification with the aspect:

```
Base       = (h_1.loadInCache -> Base).
||Woven    = (Base || Aspect).
```

Such a formalization enables us to check for properties with LTSA. For instance, the following property `P` states that every cache loading on the first host is executed locally, then replicated on the second host.

```
property P = (h_1.loadInCache -> h_1.proceed -> h_2.loadInCache -> P).
||Check_P  = (Woven || P).
```

In fact, the property `P` is violated by our system that allows the execution trace:

```
h_1.loadInCache
h_1.proceed
h_2.h_1.remoteAdvGo
h_3.h_1.remoteAdvGo
h_1.loadInCache
```

Indeed, the second occurrence of `loadInCache` can happen before the first one has been replicated on the second host because remote advices are asynchronous. We can make synchronous the remote advices by adding an event `remoteAdvEnd` at the end of each remote advice, and by waiting for these events in the local aspect:

```
Local = (h_1.loadInCache -> h_1.proceed ->
          h_2.h_1.remoteAdvGo -> h_3.h_1.remoteAdvGo ->
            h_2.h_1.remoteAdvEnd -> h_3.h_1.remoteAdvEnd -> Local).
```

In this case, the property `P` is satisfied.

The generic translation from the AWED aspect to LTS is defined in Fig. 7. We use a `for all` notation in order to enumerate hosts in each group. The translation generalizes the previous example by introducing indexes (*e.g.*, at line 6, `Local_i_j` is indexed by `i` from `G_i`) in order to avoid spurious synchronization.

4.2 Formalization of Invasive Patterns

Invasive patterns introduce two more notions. First, an invasive pattern defines a sequence of aspects. Initially, only the first aspect of the sequence is active. Once this first aspect has crosscut, and its remote advice is executed, then the second aspect becomes active. And so on. This is formalized by adding a waiting loop at the beginning of each aspect in order to ignore the joinpoint when the aspect is not active. Moreover, each aspect in a sequence of aspects, waits for an event `step_`i to become active and its remote advice generates an event `step_`$i+1$ to activate the next aspect, where i is the rank of the aspect in the sequence. Second, quite similarly to AWED, each aspect in an invasive pattern relates two groups of hosts `G1` and `G2`, but each aspect in an invasive pattern requires a rendez-vous in order to execute a remote advice. Indeed, a remote advice of an

```
// semantics of
// G_i.evt LADV G_{i+1}.RADV

// a full aspect i
||Aspect_i = (
  (for all h_j in G_i. ||Local_i_j)
  ||
  (for all h_k in G_{i+1}.
    || RemoteAdvice_i_k)
).

// local aspect (and advice)
// to de deployed on h_j in G_i
Local_i_j = (









  // pointcut
  h_j.evt ->
  // local advice
  h_j.LADV_i ->
  // start remote advices
  for all h_k in G_{i+1}.
    h_k.h_j.remoteAdvGo_i -> ...
  // wait for end of remote advices
  // only if synch remote advice
  for all h_k in G_{i+1}.
    h_k.h_j.remoteAdvEnd_i -> ...
  // and start again
  Local_i_j

).

// remote advice to be deployed
// on h_k in G_{i+1}
RemoteAdvice_i_k = (
  // wait for start by any h_j
  for all h_j in G_i.
    | (h_k.h_j.remoteAdvGo_i ->
       // remote advice
       h_k.RADV_i ->
       // end of synch remote advice

       h_k.h_j.remoteAdvEnd_i ->


       // and start again
       RemoteAdvice_i_k)
).
```

Fig. 7. Semantics of AWED Aspects

```
// semantics of
// G_i evt LADV RADV G_{i+1}

// a full aspect
||Aspect_i = (
  (for all h_j in G_i.||Local_i_j)
  ||
  (for all h_k in G_{i+1}.
    || RemoteAdvice_i_k)
).

// local aspect (and advice)
// to de deployed on h_j in G_i
Local_i_j = (
* // waiting loop
* h_j.evt -> Local_i_j
* |
* // my turn
* h_j.step_i -> (
    // pointcut
    h_j.evt ->
    // local advice
    h_j.LADV_i ->
    // start remote advices
    for all h_k in G_{i+1}.
      h_k.h_j.remoteAdvGo_i -> ...
    // wait for end of remote adv.
    // only if synch remote advice
    for all h_k in G_{i+1}.
      h_k.h_j.remoteAdvEnd_i -> ...
    // and start again
    Local_i_j
* )
).

// remote advice to be deployed
//on h_k in G_{i+1}
RemoteAdvice_i_k = (
* // rendez vous
* for all h_j in G_i.
*   h_k.h_j.remoteAdvGo_i -> ...
  // remote advice
  h_k.RADV_i ->
  // end of synch remote advice
* for all h_j in G_i.
    h_k.h_j.remoteAdvEnd_i -> ...
* // activate the next aspect
* h_k.step_{i+1} ->
  // and start again
  RemoteAdvice_i_k
).
```

Fig. 8. Semantics of invasive patterns

invasive patterns is executed on a host of G2 only when a pointcut has been detected on *each* host of G1.

The formalization of an invasive pattern from the group G_i to the group G_i+1 is defined in Fig. 8. It is an extension of the AWED semantics in Fig. 7: the lines marked with a star * are new lines. First, each aspect starts by a waiting loop in order to ignore joinpoint when the aspect is not active (lines 15 and 16). Line 17 specifies a choice noted | with a second branch: an aspect becomes active, when the right step of the sequence has been reached (lines 18 and 19). A remote advice starts now by a rendez-vous of all hosts of the group G_i (lines 39-41). The next step of the sequence is activated (with respect to the current remote host h_k) only when the remote advice is over (lines 47 and 48).

This formalization can be used to (model) check properties as in the previous section. For example, consider again a replicated cache where the following invasive pattern implements a distributed two phases commit protocol with three aspects:

```
G_a prepare { proceed } { prepareR } ;
G_b true    { }         { }          ;
G_a true    { commit }  { commitR }  G_b
where
G_a = {h_1}, G_b = {h_2,h_3}
```

The two phases of the protocol are: prepare then commit. Both phases must be replicated on remote hosts. The first aspect replicates the first phase: it crosscuts prepare on G_a (*i.e.*, h_1), performs locally this function and its remote advice performs prepareR on G_b (*i.e.*, h_2 and h_3) and activates the second aspect. The second aspect synchronizes the end of the first phase and the beginning of the second phase on different groups (G_b and G_a): it does not wait for a particular joinpoint (*i.e.*, its pointcut is true) and it does not perform a particular action (its advices are empty) but as soon as it has been activated it activates the third aspect on G_a. Finally, the third aspect performs the second phase: it does not wait for a particular joinpoint (*i.e.*, its pointcut is true) but as soon as it has been activated it performs commit locally on G_a and commitR remotely on G_b.

The formalization of this example in LTS can be checked with LTSA. For instance, the following property verifies that the two phases on h_1 are replicated on h_2:

```
P2 = (h_1.prepare-> h_2.prepareR_1-> h_1.commit_3-> h_2.commitR_3-> P2).
```

In fact the semantics does not satisfy this property. Indeed, there is a waiting loop at the start of the first aspect that can ignore an arbitrary number of joinpoints. For instance, the short trace

```
h_1.prepare
h_1.step
```

leads to a progress violation, since the first aspect is initially inactive and it can ignore the first occurrence of prepare. So, in this case this first event is not replicated (it does not trigger h_2.prepareR_1). In order to take this into account, the property must be slightly modified as follows:

```
P2 = ( h_1.prepare -> P2
     | h_2.prepareR_1 -> h_1.commit_3 -> h_2.commitR_3 -> P2).
```

An alternative option would be to remove the waiting loop at the start of the first aspect in order to take into account all occurrences of the first phase.

Our semantics can also be used to check invasive pattern definition equivalence. For instance, the previous invasive pattern seems equivalent to

```
G_a prepare { proceed  } { }         ;
G_b true    { prepareR } { }         ;
G_a true    { commit }   { commitR } G_b
```

The remote advice `prepareR` of the first aspect can be slightly delayed by shifting its code to the local advice of the second aspect. Indeed, the pointcut of the second aspect is `true`. So its local advice is executed as soon as the aspect is activated. However this is not equivalent: the second aspect is activated as soon as all remote advices of the first aspect are over (*i.e.*, there is a rendez-vous at the end of remote advices). So, our original definition enforces that all `prepareR` are over before the protocol proceed, while this alternative definition does not. We discovered such a difference while checking for other properties with LTSA.

5 Related Work

Event-based AOP [5, 8] is a general framework for AOP where pointcuts relate sequences of events. The framework is defined formally, allowing to automatically deduce possible malicious interactions between aspects. The interaction analysis remains applicable to Invasive Patterns that could be considered as a subset of EAOP. Tracematches [9] extends AspectJ with aspect-like constructs to detect sequences of joinpoints with free variables. The interaction of several tracematches is determined by the *precedence* mechanism of AspectJ. Our formalism allows a fine grained control over aspect precedence and pattern composition, providing also a model to detect sequence of joinpoints in distributed settings. Concurrent Event-based AOP (CEAOP) [10] uses LTS(A) to model base program, concurrent aspects, and their interactions. Our use of LTS(A) is analogous, but with a special focus on distribution. Caromel *et. al.* introduce a concurrent object calculus [11] implemented by the Java library ProActive [12]. Different from LTS, asynchronous communication and *futures* is at the root of this calculus. Katz's work [13] and Djoko's work [14] classify aspects in terms of the safety and liveness properties of the base program that are preserved after weaving a single type of aspect. We contribute in the verification of safety and liveness properties that are the result of complex interaction between several aspects in a distributed setting.

6 Conclusion

In previous publications we have presented AWED and invasive patterns. This paper specifies their semantics using labeled transition systems. It clarifies the

different interactions that these aspect languages introduce. Thus, it allows the verification of safety and liveness properties of the woven program, such as *deadlock freedom* and *progress*, using the LTSA tool. This paper is a step towards a complete specification that should consider specific aspect-based concurrency constructs. In this regards, an integration with CEAOP is seen as future work.

References

1. Benavides Navarro, L.D., Südholt, M., Douence, R., Menaud, J.M.: Invasive patterns for distributed programs. In: Proc. of the 9th Int. Symp. on Dist. Obj., Midd., and App. (DOA'07), Springer Verlag (November 2007) 772–789
2. Benavides Navarro, L.D., Südholt, M., Vanderperren, W., Fraine, B.D., Suvée, D.: Explicitly distributed aop using awed. In: AOSD '06: Proc. of the 5th international Conf. on Aspect-oriented software development, New York, NY, USA, ACM Press (2006) 51–62
3. Magee, J., Kramer, J.: Concurrency: State Models and Java. 2nd edn. Wiley (2006)
4. Benavides Navarro, L.D., Südholt, M., Vanderperren, W., Verheecke, B.: Modularization of distributed web services using awed. In: Proc. of the 8th Int. Conf. on Distributed Objects and Applications. LNCS 4276, Springer Verlag (October 2006) 1449–1466
5. Douence, R., Fradet, P., Südholt, M.: A framework for the detection and resolution of aspect interactions. In: Proc. of GPCE'02. Volume 2487 of LLNCS., Springer-Verlag (October 2002) 173–188
6. AspectJ: Aspectj home page. http://www.eclipse.org/aspectj (2008)
7. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of aspectj. In: ECOOP '01: Proceedings of the 15th European Conf. on Object-Oriented Programming. Springer-Verlag, London, UK (2001) 327–353
8. Douence, R., Fradet, P., Südholt, M.: Composition, reuse and interaction analysis of stateful aspects. In: Aspect-Oriented Software Development (AOSD04), ACM, ACM Press (March 2004) 141–150
9. Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Adding trace matching with free variables to aspectj. SIGPLAN Not. **40**(10) (2005) 345–364
10. Douence, R., Botlan, D.L., Noyé, J., Südholt, M.: Concurrent aspects. In: GPCE '06: Proc. of the 5th Int. Conf. on Generative programming and component engineering, New York, NY, USA, ACM (2006) 79–88
11. Caromel, D., Henrio, L., Serpette, B.P.: Asynchronous and deterministic objects. SIGPLAN Not. **39**(1) (2004) 123–134
12. Baduel, L., Baude, F., Caromel, D., Contes, A., Huet, F., Morel, M., Quilici, R.: Programming, deploying, composing, for the grid. In Cunha, J.C., Rana, O.F., eds.: Grid Computing: Software Environments and Tools. Springer-Verlag (January 2006)
13. Katz, S.: Aspect categories and classes of temporal properties. Trans. on Aspect Oriented Software Development (TAOSD) (2006) 106–134 LNCS 3880.
14. Djoko, S.D., Douence, R., Fradet, P.: Aspects preserving properties. In: ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'08), San Francisco, CA, USA, ACM Press (January 2008) 135–145