

# VPA-based aspects: better support for AOP over protocols<sup>\*</sup>

Dong Ha Nguyen and Mario Südholt  
OBASCO project

Ecole des Mines de Nantes - INRIA, LINA  
4 rue Alfred Kastler

44307 Nantes cedex 3, France

{Ha.Nguyen, Mario.Sudholt}@emn.fr

## Abstract

*Aspect-Oriented Programming is a promising approach to the construction of large-scale software systems. The declarativeness of aspect definitions and support for verification of AO programs crucially depends on the expressiveness of the aspect languages used. Currently, a large spectrum of pointcut languages, i.e., the languages that define where aspects may apply modifications to an application, have been proposed. Their expressiveness ranges from regular expression languages, which, e.g., provide support for static interaction analysis, to context-free or turing complete languages, the latter almost without any support for analysis or verification. In this paper we investigate the use of Visibly Pushdown Automata (VPA) [4] as a basis for an aspect language in order to enable more declarative aspect definitions (compared to regular approaches) for protocol-like relationships and static verification of properties, in particular analysis of interactions among aspects.*

*Concretely, we present four contributions: (i) we provide a set of examples motivating the use of VPA-based aspect definitions in the context of P2P systems, (ii) formally define a core aspect language for protocols with a VPA-based pointcut language, (iii) show that this language supports the analysis of interaction properties among aspects, and (iv) briefly present a freely available library implementing basic VPA operations, which we have used to analyze some interaction examples.*

## 1 Introduction

Aspect-Oriented Programming (AOP) [16, 1] is a research domain that strives for new modularization techniques for so-called crosscutting concerns, that is, func-

tionalties that cannot be satisfactorily encapsulated using traditional means for modularization, such as components and objects. Crosscutting concerns abound, in particular, in large-scale component-based systems (as recently shown, e.g., for data replication and transaction code in the JBoss distributed application server [6]).

By now a large number of AOP systems have been proposed for component-based systems. However, most of these approaches (see, e.g., [21, 20, 25, 15]) use turing-complete pointcut languages for the definition of the execution events where an aspect should modify the base system. This generality hinders the declarative definition of aspects and does not allow the static analysis or verification of properties over aspects.

It is well-known in the field of component-based software development that protocols are useful to support as well declarative programmatic access to components as automatic reasoning about component properties. This feature has been realized by the many approaches using finite-state protocols (see, e.g., [28, 3, 22, 13]) and some using more expressive non-regular protocols (e.g., [7, 24, 19]). In contrast, there are only few AOP systems that exploit protocol-based pointcut languages in order to enable declarative aspect definitions and provide support for reasoning over properties of AO programs. A few regular pointcut languages have been proposed (most notably [8, 9, 2]) and very few non-regular but not turing-complete pointcut languages (e.g., [27]), have been considered; no property support or even a reusable implementation has been put forward for the approaches of the latter class. In this paper, we present a new approach to AOP based on a class of non-regular protocols. The class of non-regular protocols we consider are those defined by visibly pushdown automata (VPA), which have recently been introduced by Alur and Madhusudan [4].

Concretely, we present four contributions. First, we motivate the usefulness of VPA-based aspects by a series of example aspects for the adaptation and optimization of a distributed P2P application. Second, we present an aspect

---

<sup>\*</sup>This work has been supported by AOSD-Europe, the European Network of Excellence in AOSD (<http://www.aosd-europe.net>).

language that features VPA-based pointcuts and, provides, in particular, constructors for the declarative definition of pointcuts based on regular structures, non-regular structures and the nesting depth of method calls. This language serves as a core language for the investigation of AO concepts over protocols and corresponding semantic definitions. Its semantics of the language is formally defined in terms of a small-step operational semantics by introducing an extension of an existing semantic framework for regular aspects to the VPA-based case. Third, we show by example how standard operations on VPA can be used to analyze interactions among VPA-based aspects. Finally, we briefly present a freely available implementation [17] of basic VPA operations we have developed and used to do initial experiments on interaction analysis for VPA-based aspects.

The paper is structured as follows. We motivate the use of VPA-based protocols for aspect definitions in Sec. 2. In Section 3, we present and formally define our aspect language for VPA-based aspects which includes explicit support for regular protocols and VPA-based protocols. We show in Section 4 how VPA-based aspects can be analyzed for interactions and present our prototype implementation of basic VPA operations we have realized our experiments in Section 5. We discuss related work in Section 6, followed by a conclusion and discussion of future work in Section 7.

## 2 Motivation

In this section we motivate the usefulness of VPA-based aspects and present some features of our proposed aspect language in the context of peer-to-peer (P2P) architectures.

Intuitively, visibly pushdown automata (VPA) are a restricted form of pushdown automata (PDA) where transitions that manipulate the stack are made explicit. Since VPA allow stack manipulations, they can be used to define many context-free languages and they are therefore more expressive than finite-state automata. However, the decision problems of universality/equivalence and inclusion are decidable for VPA. Hence, VPA support static analysis of properties of protocols much as regular languages do but unlike context-free languages.

P2P applications are distributed applications which are characterized by the importance of scalability and self-organization properties because of their typically very large user base, and the use of unstable and often low-bandwidth connections on the client but also server side [23]. The need for scalability and support for reorganization has led to the development of a large number of algorithms using P2P-specific protocols, among others, for searching of files and trust management in such networks.

In the following we present a number of (small-scale) example protocols and aspects for P2P applications where

VPA-based relationships are more useful than regular protocols.

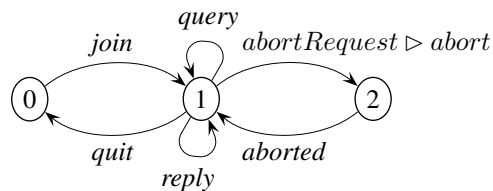


Figure 1. Aborting on-going queries (regular)

*Aborting on-going queries.* The basic advantage of VPA-based aspects over regular aspects can be illustrated by means of a protocol to abort on-going recursive queries. Fig. 1 shows a protocol which allows nodes to join or quit a P2P network. Once a node has joined, it can initiate queries and receive replies from queries. Furthermore, if an abort request occurs all queries should be aborted and new queries only be possible after all previous ones have been aborted. With aspects, abort requests can be served by triggering an advice *abort* on the occurrence of the execution event *abortRequest*: we note such a basic aspect as *abortRequest > abort*. The (complete) regular aspect shown in the figure allows these interactions; however, it does not enforce the restriction that abort requests should only be allowed if there is at least one on-going query. This is a reasonable constraint if abortion is an operation which might consume much time or other resources. Moreover, the protocol in Fig. 1 also allows the number of replies occurring at state 1 to equal (or perhaps even exceed) the number of queries.

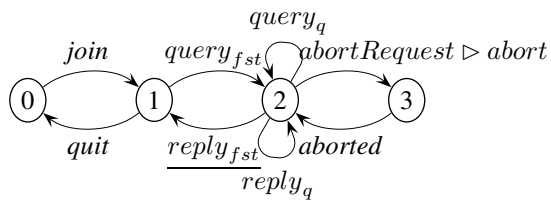


Figure 2. Aborting on-going queries (VPA)

In contrast to finite-state automata, VPA have a stack which can be used to distinguish different calls to the same method. This way, the constraint of allowing abort requests only if there are on-going queries can be expressed, e.g., using the VPA shown in Fig. 2. The main characteristic of VPA is that calls and returns are distinguished, respectively push and pop symbols of a stack, and a return transition can only be performed if the symbol of a matching call is on top of the stack. (Here and in the following call and return operations are indexed with symbols which are pushed or popped on the VPA's stack; furthermore return operations are underlined.) In the VPA example in Fig. 2, the

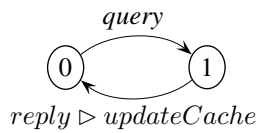
stack symbols allow to distinguish the first query operation from the remaining ones and the VPA behavior ensures that in state 2 only a number of replies can be performed that equals the number of queries done *at state 2* (i.e., with index  $q$ ), while the query with index  $fst$  performed between states 1 and 2 remains on the stack: therefore in state 2 there is always at least one query on the stack and abort requests are always made in an appropriate state.

Basically, the previous example illustrates that VPA therefore allow to “count” calls and allow to require that the number of returns match the number of calls. Finite-state automata can express similar relationships only if the maximum number of matched calls and returns is fixed (because a finite number of states can then be used to enumerate all possible cases).

*Cache updates with cut-off heuristics.* As a second example, let us consider that we want to update cache information at a node  $n$  only for those files which have been found in the vicinity of node  $N$ , i.e., at occurrence of a *reply* at a distance smaller or equal  $k$ . Using VPA we can define (the definition is given later, see Fig. 7) an operator  $D_c^{\leq k}$  matching all  $c$  occurring at depth smaller or equal to  $k$  and ignoring arbitrarily more deeply nested occurrences (this is not possible with finite-state automata). We can therefore define our cache update aspect as follows:

$$\mu a. D_{reply}^{\leq k} \triangleright updateCache(N); a$$

In this example we use a recursion introduced by  $\mu$  over the recursion variable  $a$  to allow repeated updates at all distances smaller or equal to  $k$ . (Note that we do not indicate stack symbols or mark calls and returns when the type of events are, as is the case here, irrelevant or obvious from the context.) The textual definition of aspects as exemplified here is equivalent to the graphical one introduced in the previous example. We will use the graphical representation mainly for illustration purposes and the textual for the language definition later on.

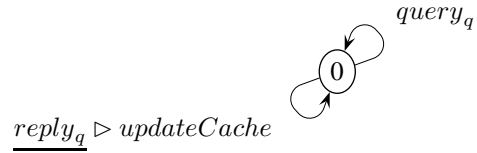


**Figure 3. Sequential queries**

*Sequential vs. parallel querying of neighbors.* As a third example, consider the protocol *Seq* defined as:

$$Seq = \mu a. query; \underline{reply} \triangleright updateCache; a$$

and shown in Fig. 3. This regular protocol allows queries to be emitted to all neighbors of a node and update the local cache at the emitting node once a reply occurs. Note, however, that queries and replies are sequentialized, a rather unusual protocol for a P2P network.



**Figure 4. Parallel queries**

A parallel query protocol can be defined as shown in Fig. 4 or alternatively as the following aspect *Par*:

$$Par = \mu a. (query_q \sqcap \underline{reply_q} \triangleright updateCache); a$$

However, this only works correctly if a VPA is used (indicated here by indexing events with stack symbols) but not with regular protocols. Once again, VPA ensure that the number of queries and replies match, while regular protocols do not ensure this restriction.

*File and trust queries.* As a last motivating example, let us consider two basic functionalities of P2P networks: file and trust queries. Trust is frequently computed by a statistical analysis of past transactions of agents in a network. After each transaction, agents may register complaints which are stored in a distributed, partially-replicated, data structure. A simple trust query may be defined as an aspect which recursively visits all neighbors and updates the data structure at the local node. This can be modeled as the following aspect:

$$Trust = \mu a. (query_q \sqcap (\underline{reply_q}; updateData)); a$$

A file query protocol may first initialize some data structure (e.g., a shared cache holding information on visited neighboring subgraphs), then attempt to lookup the file locally and if the file is not present locally initiate remote queries. Furthermore, remote queries may be ordered depending on the shared data about previous queries (this information may be useful not to execute queries immediately which involve parts of the network which are very difficult to access or which are not trusted by the current node). A suitable aspect may be defined as follows (see also Fig. 5):

$$File = \begin{aligned} &init; \mu a. lookup; (found; a \\ &\quad \sqcap fixOrder; \mu b. (query_q; b \\ &\quad \quad \sqcap \underline{reply_q}; b \\ &\quad \quad \sqcap updateData \\ &\quad \quad \quad \triangleright fixOrder; b \\ &\quad \quad \sqcap queryEnded; a)) \end{aligned}$$

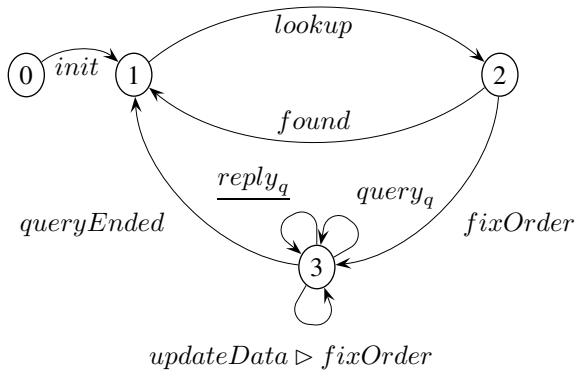


Figure 5. File query aspect

### 3 A language for VPA-based protocol aspects

We now define our language for VPA-based aspects. We first give some background on visibly pushdown automata (essentially relying on [4] for this purpose) and introduce some notation which is used in the language definition, then define the syntax of VPA-based aspects, and finally formally define its semantics by extending a semantic framework for the definition of regular aspect languages.

#### 3.1 Background information on VPA

Technically, the input alphabet of a visibly pushdown automaton is partitioned into three sets:  $\Sigma_c, \Sigma_r, \Sigma_l$  where  $\Sigma_c$  is a finite set of *calls*,  $\Sigma_r$  is a finite set of *returns* and  $\Sigma_l$  is a finite set of *local actions*. A visibly pushdown automaton is a pushdown automaton which is restricted based on these partitions: it pushes onto the stack only at *calls*, it pops the stack only at *returns* and does not change the stack at *local actions*. Using this partitioning, VPA are then defined as follows.

*Definition.* A (non-deterministic) visibly pushdown automaton  $M$  on finite words over  $\langle \Sigma_c, \Sigma_r, \Sigma_l \rangle$  is a tuple

$$M = (Q, Q_{in}, \Gamma, \delta, Q_F) \quad (1)$$

where  $Q$  is a finite set of states,  $Q_{in} \subseteq Q$  is a set of initial states,  $\Gamma$  is a finite stack alphabet that contains a special bottom-of-stack symbol  $\perp$ ,  $\delta$  is a finite set of transitions consisting of three distinct types of call, local and return transitions, and  $Q_F \subseteq Q$  is a set of final states.

VPA allow many context-free but non-regular languages to be defined, e.g., the language  $\{a^n b^n \mid n \geq 0\}$ . Note that this is an example of a “counting” property ( $a$  occurs the same number of times as  $b$ ).

An interesting characteristic of VPA is that the visibly pushdown languages are closed under union, intersection, complement, concatenation and the Kleene-star operation (all like regular languages, but unlike context-free

$A$	$::=$	$\mu a.A$	
		$P \triangleright Ad$	$; A$
		$P \triangleright Ad$	$; a$
		$A \square A$	
$P$	$::=$	$T$	$  D$
		$P.P$	$  P_1 \parallel P_2$
		$P\{\text{int}\}$	$  P^+$
			$  P^*$
$D$	$::=$	$D_M^{\text{int}}$	$  D_M^{\leq \text{int}}$
$T$	$::=$	$Tl$	$  !T$
			$  T \text{ and } T$
$Tl$	$::=$	$M$	$  M_{Id}$
			$  \underline{M}_{Id}$
$Ad$	$::=$	$send(M, Id)$	
$Pr$	$::=$	$P$	// VPA-based protocol
$M$	$::=$	$const$	$  var$
			// method names
$Id$	$::=$	$const$	$  var$
			// stack, component ids

Figure 6. Syntax of aspects over VPA-based protocols

ones, which are not closed, in particular, under intersection and complement). Furthermore, the decision problems of emptiness, universality/equivalence, and inclusion are all decidable for VPA (as for non-deterministic finite-state automata, albeit with higher time-complexity, but unlike general pushdown automata, for which universality/equivalence and inclusion are undecidable).

#### 3.2 Syntax of VPA-based aspects

Figure 6 defines the syntax of aspects over VPA-based protocols that we propose. Note that this language is mainly intended to be simple (in order to support formal definition of semantics and reasoning about it); we have, however, decided to include explicit support for regular structures, mainly because it is useful for the expression of concrete applications of the formal framework.

An aspect over VPA-based protocols (non-terminal  $A$ ) can be defined using three operators: repetition ( $\mu$ ), sequencing of basic rules ( $;$ ), and a choice operation of aspects. These aspects define VPA-based expressions over basic rules  $P \triangleright Ad$  where  $P$  (a “pointcut”) represents a pattern matching join points, such as service requests in a protocol, and  $Ad$  an advice. In this paper we consider advice only which consists simply of a call to a service of a distant component with identity  $Id$ . A pointcut is composed of terms  $T$  which denote method calls or returns (or conjunctions and complement sets thereof) built using term labels  $Tl$  (which explicitly index calls  $M_{Id}$  and returns  $\underline{M}_{Id}$

with stack symbols, local calls  $M$  are unmarked), and two primitive protocols  $D$  defining constraints in terms of the nesting depth of calls.

The language defined above extends previous work on aspects over protocols in three respects:

- VPA-based pointcuts: term labels not only specify names of methods relevant to a join point but also the type of the join point (call, return, local), in particular, to enable specification of matching calls and returns.
- Two specific “depth-defining” VPA protocol constructors (see non-terminal  $D$ ):  $D_c^n$ , a protocol requiring  $n$  nested calls to  $c$ , and  $D_c^{\leq n}$ , which requires calls to  $c$  nested until depth  $n$ .
- A syntax for the definition of regular expression like pointcuts (see non-terminal  $P$ ).

### 3.3 Semantics

We now define the semantics of our aspect language for finite-state protocols in terms of an operational small-step semantics similar to the framework for regular aspects introduced by Douence *et al.* [8]. Concretely, three issues must be addressed:

- Join points: we have to define suitable transition relations defining the execution points at which aspects can be applied to an execution of a protocol.
- Pointcuts: matching of pointcuts must take VPA transitions into account. Depth-defining and regular pointcuts must be defined.
- Aspect weaving: when do pointcuts match join points and how does advice modify protocol state?

However, we can simplify the definition of the semantics compared to [8] by exploiting the observation that the sequences of pointcuts defined by an aspect (*i.e.*, without considering the interleaved advices) can be represented as one (generally non-deterministic) VPA. This VPA, henceforth called “pointcut VPA” and denoted  $PVPA^A$  for aspect  $A$ , can easily be constructed bottom up by building VPA having individual transitions for each term label generated by  $T$ , and sequencing them as well as constructing loops according to the three aspect composition operators  $\mu, ;, \square$ , cf. the rule for non-terminal  $A$  in Fig.6. In fact, as shown below we do not have to treat depth-defining and regular expression like pointcuts explicitly, because they can be reduced to aspect expressions, *i.e.*, terms generated by non-terminal  $A$  (cf. Figures 8 and 7). (Due to lack of space, we do not give the straightforward definition of this VPA, see [18] instead.)

$$D_m^n \triangleright I; A := m_c . (MCR . m_c)\{n - 1\} \triangleright i; A$$

$$\text{where } MCR := (m_c . \underline{m_c})^*$$

$$D_m^{\leq n} \triangleright I; A := \bigsqcup_{1 \leq i \leq n} (D_m^i \triangleright I; A)$$

**Figure 7. Translation of depth-defining pointcuts into regular expression pointcuts**

$$(P_1 . P_2) \triangleright I; A := P_1 \triangleright \text{skip}; P_2 \triangleright I; A$$

$$(P_1 \parallel P_2) \triangleright I; A := (P_1 \triangleright I; A) \square (P_2 \triangleright I; A)$$

$$P^* \triangleright I; A := \mu x. ((P \triangleright \text{skip}; x) \square (\text{true} \triangleright I; A))$$

**Figure 8. Translation of regular expression pointcuts into aspects**

In the following we will present the semantics of our aspect language, *i.e.*, join points, pointcuts, advice, and aspect weaving in turn.

**Join points and base transition relation.** The semantics is defined based on a small-step transition relation of observable execution steps of the base program. This relation, denoted  $\rightarrow$  relates configurations of the form  $(j, p)$ , where  $j$  is a join point and  $p$  some (dynamically evolving) program state:

$$(j, p) \rightarrow (j', p')$$

In this paper, we use this state to store the effects of advice, *i.e.*, calls to remote components. Furthermore, as usual we denote the transitive reflexive closure of a relation by indexing it with ‘\*’.

**Pointcuts.** Pointcuts (cf. non-terminal  $P$  in Fig 6) are built from terms, depth-defining protocol constructors and regular expression like pointcuts. The semantics, *i.e.*, matching, of terms (which includes matching calls and corresponding returns) will be defined as part of the weaver definition below. We now define the two remaining pointcut kinds by a reduction into aspect expressions, *i.e.*, expressions generated by non-terminal  $A$  in Fig. 6.

First, depth-defining constructors can be reduced to regular expression like pointcuts as defined in Fig. 7. A pointcut  $D_m^n$  is translated into a sequence of  $n$  calls to  $m$  interleaved with nested matching sequences of calls and returns to  $m$  (matched by the VPA  $MCR$ ). Note that in contrast to regular protocols, VPA allow here to match arbitrarily deeply nested structures. A pointcut  $D_m^{\leq n}$  is reduced to

an  $n$ -fold choice each branch of which consists of a depth-defining pointcut for one depth level.

Second, the regular expression like pointcuts generated by non-terminal  $P$  (as well as depth-defining pointcuts) are reduced to aspect expressions as shown in Figure 8. Intuitively, such pointcuts can be emulated in the original framework by sequences of basic rules whose advices, except that of the last rule in the sequence, do not modify the protocol state. Formally, the semantics is given by a transformation into regular aspects of the original framework (generated by non-terminal  $A$  in Fig. 6). In the figure, `skip` represents the “empty” advice. (The rules for the regular operators  $+$  and  $\{int\}$  are not shown but trivial to define based on the given ones.)

**Advice.** Our advice language (non-terminal  $Ad$  in Fig. 6) only consists in calls to services of remote components. We define its semantics simply by recording the trace of corresponding service calls, *i.e.*, the semantics of advice  $send(m, id)$  called as part of an aspect in state  $(j, p)$  is defined by:

$$(j, p) \rightarrow (j, p ++ (m, id))$$

Note that advice execution is modeled using an advice transition relation ( $\rightarrow$ ) that may be different from the base transition relation. This is useful, *e.g.*, to exempt advice from weaving (Because join points are generated only by executions defined using the base transition relation).

**Aspect weaving.** Aspect weaving of our aspect language requires the following issues to be defined:

1. Matching of terms  $T$  (conjunctions of term labels and complement thereof). Matching also takes into account the handling of matching calls and returns, the basic property differentiating VPA from finite-state automata.
2. Which aspects are applicable at a join point (since several aspects may be applied at once) and how to compute the follow state of the applicable aspect.
3. How applicable aspects are woven and the program state is modified by the aspect application.

*Matching of terms.* Matching (henceforth denoted by **match**  $j s$  where  $j$  is the current join point and  $s$  denotes the current state of the pointcut VPA) performs three steps. First, the transition of the pointcut VPA which match against the current join point are determined (see [8] for details). Second, for all matching return transitions, we have to test whether a matching call is on top the current stack of the pointcut VPA. Third, a variable assignment is returned which binds the variables used to pass information

from pointcuts to advice (cf. non-terminal  $M$  in Fig. 6; this assignment is empty if the match has not been successful).

*Selection of applicable aspects and follow states.* To define applicability of aspects, first note that because of non-determinism an aspect may have different current states. We first define the function **sel** which given a join point  $j$  and a set of current states  $ss$  (*i.e.*, a subset of the state set of  $PVPA^A$ ) yields all applicable pointcuts (which must be terms since more complex pointcuts have been reduced) and associated advice (which may be empty):

$$\mathbf{sel} j ss = \{(p_s, a_s) \mid s \in ss, \mathbf{match} j s\}$$

where  $p_s$  and  $a_s$  respectively denote the pointcut and advice corresponding to state  $s$  of  $PVPA^A$ .

As to the follow states of an aspect after weaving at a join point, note that the aspect may remain in the same state as before (in case none of the pointcuts associated to the current states matched the join point) or evolve to a new state. We define this using the function **next** which takes a composite aspect and yields the aspect to be applied to the next join point, *i.e.*, defined as:

$$\begin{aligned} \mathbf{next} j ss &= \{s \in ss \mid \mathbf{not} \mathbf{match} j p_s\} \\ &\cup \{s' \mid (s, -, -, s') \in \delta, \mathbf{match} j p_s\} \end{aligned}$$

where  $(s, -, -, s')$  matches any transition in  $\delta$  irrespective of its type (call, return, local).

*Aspect weaving.* We now define aspect weaving based on the base and advice transition relations ( $\rightarrow, \rightarrow$ ), a set of current states  $ss$  and the advice mapping *advice*. In Figure 9, weaving is decomposed in two steps. First, the weaving step proper (the topmost rule) which takes an aspect and applies it to a join point: this step selects all applicable states from the aspect, applies it to the current join point, thus producing a new component state  $p'$ . One step of the base program is then performed to produce a new join point (which possibly also modifies the protocol state). Finally, the aspect evolves using the function **next** and the new aspect state, join point and protocol state are returned. Second, the two bottom-most aspect application rules define how the current set of applicable aspect states are woven: the middle rule terminates aspect application when all aspect states have been woven; the bottom rule weaves one applicable aspect state  $s$  by determining possible variable assignments when the pointcut  $p_s$  is matched on the current join point. Finally, the rule states that the corresponding advice  $a_s$  is executed using the advice transition relation from start to end at join point  $j$ , thus producing a new protocol state  $p'$ .

Note that the core aspect language we have introduced is not intended for programming at the user-level. A more user-friendly aspect language should integrate the core language into a more mainstream aspect language. We are considering a user-level language based on the AWED aspect

$$\frac{[j, p] \text{sel } j \text{ ss} \xrightarrow{*} p' \quad (j, p') \rightarrow (j', p'')}{(ss, j, p) \Rightarrow (\text{next } j \text{ ss}, j', p'')}$$

Aspect application rules

$$[j, p]^0 \Longrightarrow p$$

$$\frac{\mathcal{S}' = \mathcal{S} - \{s\} \quad \mathcal{S} \neq \mathcal{S}' \quad \text{match } j \text{ p}_s = \phi \quad (\text{start}, \phi a_s, p) \xrightarrow{*} (\text{end}, \phi a_s, p')}{[j, p]^{\mathcal{S}} \Longrightarrow [j, p']^{\mathcal{S}'}}$$

Figure 9. Weaving aspects on VPA-based protocols

language [6]. As an example, Fig. 10 presents the aspect *Par* using this syntax: note that the terms `callC`, `callR` allow to match two operations via a stack symbol given as the second argument, and the sequence construct allows to define non-deterministic automata transitions.

```

aspect ParallelQuery{
  pointcut ParQuery:
    Par: seq( s0: callC(query, q) → s0 || s1,
             s1: callR(reply, q) → s0 || s1)
  after: step(Par,s1){ updateCache(); } }

```

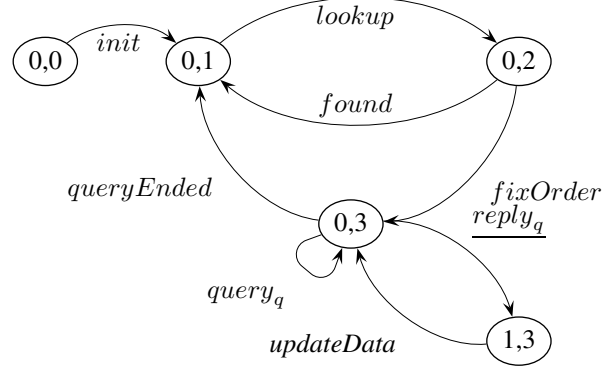
Figure 10. *Par* aspect expressed in user-friendly syntax

Figure 11. Product of Trust and File query aspects

#### 4 Interaction analysis for VPA-based aspects

The second salient characteristic of VPA — besides its increased expressiveness compared to finite-state automata, which we have exploited in our motivating examples — is that they enjoy the same closure properties as finite-state automata. It therefore stands to reason that the interaction analysis among aspects introduced for the regular case by Douence *et al.* [8, 10] should be generalizable to VPA.

To make this discussion more concrete, reconsider the file and trust query example aspects (denoted as *File* and *Trust* in Sec. 2, respectively). Note that both may update the shared data structure. Furthermore, on an update event the file query aspect may change the order in which subsequent queries are done. Intuitively, since both may update the shared data structure, they may interact if both are applied at the same time (which is the case in P2P networks where both types of queries are typically executed in parallel). Now imagine that both aspects trigger an update at the same time and that the trust query returns a result that the searched subgraph should be marked as a “cheater”: in this

case the update to be initiated by the file query should probably be ignored and all queries that are still on-going and that are performed within the realm of the cheater should probably be cancelled.

Using basic operations on VPA and their closure properties, interactions in the sense of simultaneous matches of the same atomic pointcut should, in principle, be statically analyzable by calculating the product automaton of two VPA-based aspects. If the resulting product automaton contains simultaneous occurrences of the same transitions in both VPA, we conclude that there are potential interactions in two corresponding VPA-based aspects.

We have employed our VPA library (described in the following section) to calculate the product of the two example query aspects introduced in the motivation section. The resulting product automaton is shown in Fig. 11. This automaton explicitly represents the interaction between both aspects on the *updateData* event. Formally, the product also indicates potential interactions on *query* and *reply*, these are, however, not interesting in the context of the example we considered, because no actions are triggered on

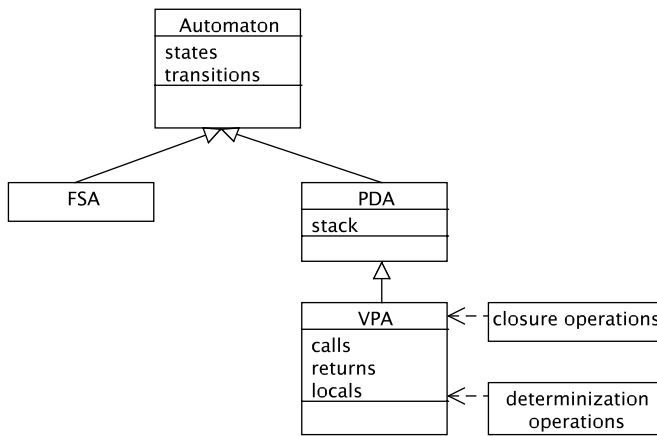


Figure 12. VPA library class diagram

these events. Note that events of the file query aspect which are unknown to the trust query aspect, such as *init* and *lookup*, are allowed to be interleaved in the resulting product automaton and do not affect the analysis result.

## 5 A VPA library

We have realized a library [17] that implements basic operations on VPA to support experimentation of VPA-based protocols. The implementation has been performed using Java 5, exploiting, in particular, its facilities for generic data types, enumeration types, and iterator. In this section we will give a short overview of the architecture of the prototype and a glimpse of its implementation.

Figure 12 shows the class hierarchy and some more important field variables of the library. The library currently supports three different types of automata: finite state automata (FSA), pushdown automata (PDA), and visibly pushdown automata (VPA).

The class VPA represents the necessary input partitioning explicitly through members *calls*, *returns* and *locals* (states, transitions are inherited from the base class Automaton, and stack from class PDA). The closure operations union, intersection and Kleene star have been implemented for VPA as part of class ClosureOperation. The class Determinization allows the construction of a deterministic VPA from a non-deterministic one.

Figure 13 shows part of our implementation of the Kleene star operation for VPA. The result VPA  $M^*$  is constructed from the VPA  $M$  by simulating  $M$  step by step and, as soon as  $M$  changes its state to a final state,  $M^*$  non-deterministically updates its state to an initial state. From that point on, the stack is treated as if it were empty and states of  $M$  are tagged to distinguish old and new states.

Our implementation of the star operation is very close to its formal definition. First the argument VPA (line 3) is

```

1 public VPA star(VPA v) {
2     // Deep copy argument to result (Q_in_res = Q_in)
3     VPA res = new VPA(v);
4
5     // Q_res = Q \cup Q^tag
6     HashSet<State> taggedStates = new HashSet<State>();
7     for (State s: v.getStates())
8         taggedStates.add(Misc.tagState(s));
9     res.addAllStates(taggedStates);
10
11    ... // Q_F_res = Q_F \cup Q_F^tag
12
13    ... // Gamma_res = Gamma \cup Gamma^tag
14
15    // delta_res
16    HashSet<Transition>
17    delta = new HashSet<Transition>(v.getTransitions());
18    VPATransition tau;
19    for (Transition at: v.getTransitions()) {
20        VPATransition t = (VPATransition) at;
21
22        State q = t.getFrom();
23        String a = t.getInput();
24        State p = t.getTo();
25
26        switch (t.trSort) {
27            case local: {
28                tau = new VPATransition(Misc.tagState(q),
29                                     a, Misc.tagState(p));
30                res.addTransition(tau);
31
32                if (v.getFinalStates().contains(p)) {
33                    for (State r: v.getInitialStates()) {
34                        tau = new VPATransition(q, a,
35                                               Misc.tagState(r));
36                        res.addTransition(tau);
37                        tau = new VPATransition(Misc.tagState(q),
38                                             a, Misc.tagState(r));
39                        res.addTransition(tau); } }
40                break;
41            }
42            case push: { ... }
43            case pop: { ... }
44        } // switch
45    }
46
47    return res; }
  
```

Figure 13. Excerpt of implementation of Kleene-star on VPA

deeply copied to include the original VPA  $v$  in the result automaton. Then, the tagged elements are added to the original components as shown for the state set  $Q$  in lines 5–9. Finally, the new transition function is calculated (lines 17–55), basically by iterating over all transitions of the original transition function (for-loop in line 21) and adding corresponding new transitions as shown in the excerpt for the case of transitions of type “local” (lines 29–47).

The current first implementation is naive especially in that it creates all data structures eagerly, *e.g.*, creating upfront an exponential number of states as part of determinization operation instead of lazily creating needed states only. Note that, however, most interesting operations, such as the test of inclusion of two visibly pushdown languages — which is the main operation needed for the analysis of

whether two components with such protocols are substitutable — can be computed in polynomial time.

## 6 Related work

The large majority of current AOP languages, foremost of all AspectJ [5], provide only inappropriate support for protocol-based aspects as considered in this paper for two reasons. First, they essentially only support atomic pointcuts, that is, pointcuts only denote sets of individual execution points but cannot directly represent relationships between execution points. AspectJ, for instance, only supports such atomic join points (with the only exception of its cflow-pointcut), relationships between execution points have to be programmed using an aspect-internal state. Second, the pointcut languages are typically turing-complete: AspectJ allows, *e.g.*, to use arbitrary conditions, implemented using arbitrary Java methods, in pointcut definitions. These languages do not support non-trivial reasoning over pointcuts and cannot support analysis of interaction properties as we have considered. In contrast, Douence *et al.* [11] have proposed a formally-defined turing-complete pointcut language, which supports manual proofs of properties of aspects.

There is a number of approaches using regular aspect languages that feature explicit notions of protocols. As mentioned [8, 10] introduced the semantic framework and interaction analysis that has been extended in the present paper to VPA-based aspects. Farías [12, 14] has considered a language for regular aspects that admits advice modifying the static structure of protocols and he has also investigated interaction properties in this setting. The approach of tracematches [2] can be seen as providing a complementary notion of regular aspects and features an alternative formal semantics. It has, however, not yet applied to interaction analysis among aspects. The notion of stateful aspects [26], which extends the JAsCo language [25], also provides a notion of regular aspects but still features a turing-complete pointcut language.

Finally, there are few approaches featuring non-regular pointcut languages (which are not turing-complete). A notable exception, are the tracecuts proposed by Walker and Vigger, which essentially provide a context-free pointcut language. However, they do not propose an effective implementation and do not consider analysis of properties of aspects.

## 7 Conclusion

In this paper we have proposed an aspect language based on visibly pushdown automata. Concretely, we have presented four contributions: (i) we have provided a set of

examples motivating the use of non-regular aspects in the context of P2P systems, (ii) formally defined an aspect language with a VPA-based pointcut language by extending a formal framework for regular aspects, (iii) shown that the language supports the analysis of interaction properties among aspects, and (iv) briefly presented a freely available library implementing basic VPA operations.

This work paves the way for different leads to future work. First, in a protocol-based setting, it is useful that advice can modify the protocol. A more powerful advice language, however, requires changes to the formal semantics and how properties can be analyzed. Second, a larger set of VPA-specific protocols (such as the depth-defining protocols) should be included. Third, the aspect language should be integrated in a real aspect language, which requires more efficient support for VPA (which is an active research domain of its own) and will probably raise new language design issues. Finally, the application of VPA-based aspects to component-based systems should be considered.

## References

- [1] Mehmet Akşit, Siobhán Clarke, Tzilla Elrad, and Robert E. Filman, editors. *Aspect-Oriented Software Development*. Addison-Wesley Professional, September 2004.
- [2] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, et al. Adding trace matching with free variables to AspectJ. In Richard P. Gabriel, editor, *ACM Conference on Object-Oriented Programming, Systems and Languages (OOPSLA)*. ACM Press, 2005.
- [3] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–49, July 1997.
- [4] Rajeev Alur and Parthasarathy Madhusudan. Visibly pushdown languages. In *Proceedings of the thirty-sixth annual ACM Symposium on Theory of Computing (STOC-04)*, pages 202–211, New York, June 13–15 2004. ACM Press.
- [5] AspectJ [home page](http://eclipse.org/aspectj). <http://eclipse.org/aspectj>.
- [6] Luis D. Benavides Navarro, Mario Südholt, Wim Vanderperren, Bruno De Fraine, and Davy Suvée. Explicitly distributed AOP using AWED. In *Proceedings of the ACM Conference on Aspect-Oriented Software Development (AOSD'06)*. ACM Press, March 2006.
- [7] Andrea Braccialia, Antonio Brogi, and Carlos Canal. A formal approach to component adaptation. *Journal of Systems and Software*, 2005.

- [8] Rémi Douence, Pascal Fradet, and Mario Südholt. A framework for the detection and resolution of aspect interactions. In *Proceedings of the ACM Conference on Generative Programming and Component Engineering (GPCE'02)*, volume 2487 of *LNCS*, pages 173–188. Springer Verlag, October 2002.
- [9] Rémi Douence, Pascal Fradet, and Mario Südholt. Composition, reuse and interaction analysis of stateful aspects. In *Proc. of 3rd International Conference on Aspect-Oriented Software Development (AOSD'04)*, pages 141–150. ACM Press, March 2004.
- [10] Rémi Douence, Pascal Fradet, and Mario Südholt. Trace-based aspects. In Mehmet Akşit, Siobhán Clarke, Tzilla Elrad, and Robert E. Filman, editors, *Aspect-Oriented Software Development*. Addison-Wesley Professional, September 2004.
- [11] Rémi Douence, Olivier Motelet, and Mario Südholt. A formal definition of crosscuts. In *Proceedings of Reflection 2001*, LNCS 2192, pages 170–186, 2001.
- [12] Andrés Farías. *Un modèle de composants avec des protocoles explicites*. PhD thesis, École des Mines de Nantes/Université de Nantes, December 2003.
- [13] Andrés Farías and Mario Südholt. On components with explicit protocols satisfying a notion of correctness by construction. In *International Symposium on Distributed Objects and Applications (DOA)*, volume 2519 of *LNCS*, pages 995–1006, 2002.
- [14] Andrés Farías and Mario Südholt. Integrating protocol aspects with software components to address dependability concerns. Technical Report 04/6/INFO, École des Mines de Nantes, November 2004.
- [15] Steffen Göbel, Christoph Pohl, Simone Röttger, and Steffen Zschaler. The COMQUAD component model — enabling dynamic selection of implementations by weaving non-functional aspects. In *Proceedings of AOSD'04*. ACM Press, 2004.
- [16] Gregor Kiczales, John Lamping, et al. Aspect-Oriented Programming. In *11th European Conference on Object-Oriented Programming*, LNCS 1241, pages 220–242. Springer Verlag, 1997.
- [17] Ha Nguyen and Mario Südholt. *VPA library*. <http://www.emn.fr/x-info/hnguyen/vpa>.
- [18] Ha Nguyen and Mario Südholt. Aspects over VPA-based protocols. Technical report, INRIA, July 2006.
- [19] Sebastian Pavel, Jacques Noyé, Pascal Poizat, and Jean-Claude Royer. Java implementation of a component model with explicit symbolic protocols. In *Proceedings of the 4th International Workshop on Software Composition (SC'05)*, volume 3628 of *LNCS*. Springer Verlag, April 2005.
- [20] Roman Pichler, Klaus Ostermann, and Mira Mezini. On aspectualizing component models. *Software – Practice and Experience*, 33(10):957–974, August 2003.
- [21] M. Pinto, L. Fuentes, and J.M. Troya. DAOP-ADL: an architecture description language for dynamic component and aspect-based development. In *Proc. of the 2nd Conf. on Generative Progr. and Component Engineering (GPCE)*, LNCS. Springer Verlag, 2003.
- [22] Frantisek Plasil and Stanislav Visnovsky. Behavior protocols for software components. *Transactions on Software Engineering*, 28(9), January 2002.
- [23] M. Ripeanu, I. Foster, and A. Iamnitchi. Mapping the Gnutella network: Properties of large-scale peer-to-peer systems and implications for system design. *IEEE Internet Computing Journal*, 6(1), Sep. 25 2002.
- [24] Mario Südholt. A model of components with non-regular protocols. In *Proceedings of the 4th International Workshop on Software Composition (SC'05)*, volume 3628 of *LNCS*. Springer Verlag, April 2005.
- [25] Davy Suvée, Wim Vanderperren, and Viviane Jonckers. JasCo; an aspect-oriented approach tailored for component-based software development. In ACM Press, editor, *Proc. of 2nd International Conference on Aspect-Oriented Software Development (AOSD'03)*, pages 21–29, March 2003.
- [26] Wim Vanderperren, Davy Suvée, Maria Augustina Cibran, and Bruno De Fraine. Stateful aspects in JAsCo. In *Proc. of the 4th Int. Workshop on Software Composition (SC'05)*, volume 3628 of *LNCS*. Springer Verlag, April 2005.
- [27] Robert J. Walker and Kevin Viggers. Implementing protocols via declarative event patterns. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE-12)*, pages 159 – 169. ACM Press, 2004.
- [28] Daniel M. Yellin and Robert E. Strom. Protocol specifications and component adaptors. *ACM Transactions of Programming Languages and Systems*, 19(2):292–333, March 1997.