Property-preserving evolution of components using VPA-based aspects

Dong Ha Nguyen and Mario Südholt

OBASCO project; EMN-INRIA, LINA
Dépt. Informatique, École des Mines de Nantes
4 rue Alfred Kastler, 44307 Nantes cédex 3, France
{Ha.Nguyen, Mario.Sudholt}@emn.fr

Abstract. Protocols that govern the interactions between software components are a popular means to support the construction of correct component-based systems. Previous studies have, however, almost exclusively focused on static component systems that are not subject to evolution. Evolution of component-based systems with explicit interaction protocols can be defined quite naturally using aspects (in the sense of AOP) that modify component protocols. A major question then is whether aspect-based evolutions preserve fundamental correctness properties, such as compatibility and substitutability relations between software components.
In this paper we discuss how such correctness properties can be proven in the presence of aspect languages that allow to match traces satisfying interaction protocols and enable limited modifications to protocols. We show how common evolutions of distributed components can be modeled using VPA-based aspects [14] and be proven correct directly in terms of properties of operators of the aspect language. We first present several extensions to an existing language for VPA-based aspects that facilitate the evolution of component systems. We then discuss different proof techniques for the preservation of composition properties of component-based systems that are subject to evolution using protocol-modifying aspects.

1 Introduction

Interaction protocols are a popular means to construct correct component-based systems and document them (see, *e.g.,* [23, 17, 8]). A major question for the evolution of component-based systems is whether evolution preserves compositional properties of these systems, in particular compatibility and substitutability of components, two fundamental notions that are typically defined in terms of subset relationships of trace sets (and sometimes failure sets) admitted by the original and evolved versions of a system [23, 15]. Currently, almost all component-based systems with interaction protocols have used finite-state protocols, includ-

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

ing all of the previously cited ones. Recently, however, approaches using more expressive non-regular protocols have been proposed [3, 19].

If software components are equipped with interaction protocols, evolution of component-based systems can frequently be expressed in a concise manner using aspects that modify executions that have to conform to such protocols. In the last couple of years different researchers have considered protocol-modifying aspect languages, notably [4, 6, 22, 14]. However, none of these approaches has explored the preservation of compositional properties of software components in the context of aspects modifying such interaction protocols. Among the very few that have touched on this question, Farías [7] proposes the extension of the language of regular aspects by protocol-modifying operators and considers proof techniques for the resulting finite-state based aspects.

We consider how compositional properties can be defined and verified in the context of the evolution of components that are equipped with a more expressive brand of interaction protocols, protocols defined in terms of Visibly Pushdown Automata (VPA) [2]. VPA allow to define protocols that include well-formed nested contexts, such as correct nesting of recursive calls to and returns from a server. VPAs are strictly more expressive than finite-state automata (which generate regular languages) but strictly less so than pushdown automata (which generate context-free languages). In contrast to finite-state based systems, VPA-based protocols allow (some) nested terms of, *e.g.,* call and returns, to be correctly matched without having to restrict the nesting depth. In contrast to pushdown automata, VP languages are closed under all basic operations, including intersection and complement, and all basic decision problems are decidable.

In this paper, we present two main contributions. First, we present four extensions to the language of VPA-based aspects [14] that are useful in the context of component evolution for distributed applications: a more general definition of sequence pointcuts, a new pointcut operator that allows nested contexts to be matched if their depths exceed a threshold, a permutation for well-balanced contexts and a new advice construct that allows to close an open nested context, *e.g.,* for error handling purposes. We motivate how these extensions can be used in the context of the evolution of software components for the implementation of distributed search algorithms typical, *e.g.,* for P2P applications. Second, we show how compatibility and substitutability properties of component-based applications can be proven if interaction protocols are subject to evolution using VPA-based aspects. We present, in particular, how some composition properties can be handled fully in terms of the properties of protocol-operators even in the presence of aspects.

The paper is structured as follows. Sec. 2 motivates using VPA-based aspects to define evolution of distributed components with explicit protocols and to verify the preservation of compositional properties for those systems. Sec. 3 presents the VPA-based aspect language and defines the four extensions to the language. Sec. 4 presents the proof techniques for the preservation of composition properties in the presence of aspects. In Sec. 5 we discuss related work before concluding in Sec. 6.

2 Motivation

The large majority of component-based systems is based on components whose interfaces define sets of (method or function) signatures that correspond to services provided by the component. Furthermore, signatures that correspond services that a component requires to be provided from other components are also frequently declared as part of component interfaces. Such interfaces do not, however, specify any information about the semantics of provided and required services. With such a notion of components, component evolution therefore has to be defined essentially on the level of component implementations and component properties, such as compatibility (that ensures that two components communicate correctly) and substitutability (that ensures that a component can be substituted without problems for another one in arbitrary usage contexts), can only be defined in terms of service signatures, *i.e.,* not including any guarantees on the effective behavior of the components.

Protocols that govern component interactions and that are explicit in component interfaces have been proposed as a means to overcome these limitations. Changes to interaction protocols allow to model component evolution in terms of modifications because they allow to enable new of forbid previously enabled communications between components. Furthermore, compatibility and substitutability properties can be formally based on the interaction protocols. Interaction protocols have mostly been defined in terms of finite-state automata (*e.g.,* [23, 17, 8]), only a few have considered more expressive non-regular protocol languages (*e.g.,* [3, 19]). None of these approaches, have considered, however, component properties in the context of dependencies involving recursive computations, which are, however, crucial to a large class of application domains.

In order to motivate the special requirements of such application domains, let us consider a quite typical representative, P2P algorithms, in particular, the approach to unstructured P2P networks proposed by Lv et al. [13]. A central proposal of this work is a particular algorithm to search a decentralized and unstructured P2P network. The basic algorithm can be summarized as follows. When a peer initiates a search to find a document, it sends a corresponding query message to (some of) its neighbors in some order. When a peer receives a query message it first checks its local store and replies to the query with the corresponding file if it is found locally. Otherwise, it forwards the request to its neighbors. To ensure termination, this search protocol makes use of a time-to-live (TTL) value in order to limit the search space. This value is decremented at each peer that is visited and the search is stopped when 0 is reached.

From a component point of view the behavior on a node we consider can be characterized by the incoming and outgoing communication events as shown in Fig. 1 that can be seen as defining its signature-based interface. The P2P algorithm most basically consists of nodes sending out `query` messages and receiving `reply` messages from its neighbors. Furthermore a node may `abort` on-going queries if requested to do so (`abortRequest`). Finally, a node may `join` or `quit` the P2P network.

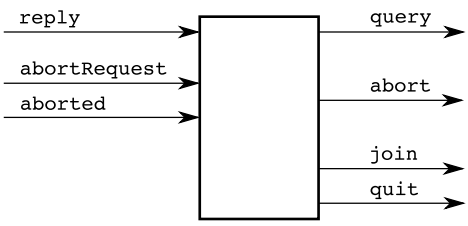


Fig. 1. Component interface (service part) for P2P algorithms

All of the (regular or non-regular) approaches to interaction protocols cited above do not permit to express (many) fundamental protocols that involve the recursive nature of the search algorithm in the above P2P context. Furthermore, none of those allow to reason about the properties underlying such protocols. VPA-based protocols allow to directly define many of such protocols and enable formal reasoning about correctness properties for components defined in terms of them. Finally, aspects over interaction protocols allow component evolution to be expressed naturally; VPA-based aspects enable evolutions to be defined that include modifications to such recursive behaviors and verify the preservation of composition properties in such cases.

To illustrate the basic advantage of VPA-based aspects over regular aspects, consider the (regular) protocol for aborting queries that is shown in Fig. 2. If an abort request occurs a query should be aborted and new queries only be enabled after abortion has been performed. With aspects, abort requests can be performed by triggering an advice *abort* when the execution event *abortRequest* occurs: we note such a basic aspect as $abortRequest \triangleright abort$. 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. 2 also allows the number of replies occurring at state 1 to exceed the number of queries (which is obviously an unreasonable situation).

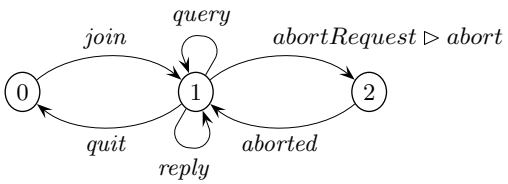


Fig. 2. Aborting on-going queries (regular)

In contrast to finite-state automata, VPAs 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. 3. The main characteristics of VPA is that calls and returns are distinguished by well-distinguished push and pop symbols on a stack, and a return transition can only be performed if the symbol of a matching call is on top of the stack. Call and return operations in the protocol are indexed with symbols which are pushed or popped on the VPA stack; furthermore return operations are underlined. In the VPA example in Fig. 3, the stack symbols allow to distinguish the first query operation from the remaining ones and the VPA behavior thus 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 example illustrates that VPA allow to "count" calls and allow to require that the number of returns match the number of calls.

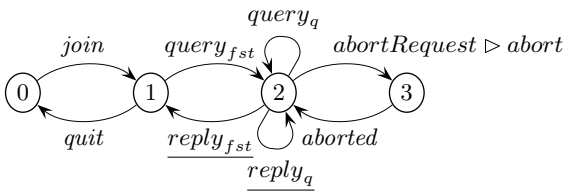


Fig. 3. Aborting on-going queries (VPA)

Finally, note that typical modifications to such recursive algorithms, such as stopping a recursive algorithm at a given call depth, cannot be directly expressed using regular structures: they can only be expressed using finite-state structures by fixing the maximum number of open recursive calls. Such a constraint is, however, (i) unpractical in application contexts such as P2P networks where the recursive depth to which a network is to be explored may be rather large and (ii) signifies that no general composition properties, *i.e.,* that do not include such limits on the recursion depth, can be proven for regular aspects. VPA-based aspects, however, provide a solution for these two problems for a large set of recursive interaction protocols.*

3 Extending VPA-based aspects for component evolution

In this section we present the VPA-based aspect language that we propose for component evolution. After presenting the main characteristics of the language by a small example, we first introduce the pointcut and advice operators that we have introduced to cope with component evolution; this part focuses on

* Since VPAs are less expressive than pushdown automata not all context-free protocols can be represented, though).

the four operators that are new compared to our previous proposal [14]. We then present the formal definition of the complete aspect language and give an informal account of the semantics of the language.

3.1 VPA-based aspects by example.

In the motivation, we have presented an aspect to `abort` search queries after an `abortRequest` message has been received. The VPA-based aspect that ensures that aborts are only performed if at least one search query is active, can be defined using our language as follows:**

$$\begin{aligned} join\ ;\ \mu b.\ query_{fst}\ ;\ \mu c.(&\underline{reply_{fst}}\ ;\ b) \\ &\Box\ (query_q \Box \underline{reply_q}\ ;\ c) \\ &\Box\ (abortRequest \triangleright abort\ ;\ c) \end{aligned}$$

As common for aspect languages, the above aspect definition essentially consists of a pointcut definition, which defines the sequences of execution events the aspect matches, and an advice definition, which defines the actions to be executed once a match occurs.

In the above aspect definition, the *pointcut* specifies sequences of events of interest for components providing services for P2P search protocol as illustrated in Fig. 1. Repetitions in a pointcut are defined using the 'μ' operator, bind a recursion variable (b in the outermost repetition above) and extend to the next occurrence of the recursion variable. Sequences of events are formed using the sequencing operator ';' (*query* events . A choice among two or more matching events is expressed by operator '$\Box$'. For instance, "$\mu c.\ query_q \ \Box\ reply_q\ ;\ c$" presents a pointcut that repeatedly matches *query* or *reply* events. Furthermore, VPA call as well as return events are indexed with stack symbols; VPA returns are underlined. For example, $query_{fst}$ and $reply_{fst}$ will match only the first query and its corresponding last reply of the search process.

The above aspect contains only one *advice*, the call to *abort* in the final branch of the choice that contains the basic aspect $abortRequest \triangleright abort$ consisting of a single event triggering the advice.

3.2 Pointcut and advice operators for component evolution

Calls and corresponding returns that are used in pointcuts of VPA-based aspects enable well-balanced contexts to be used in interaction protocols, matched during program execution and statically analyzed, *e.g.,* for conflicts with other VPA-based aspects. Such aspects may be concisely defined in terms of operators that are specialized to well-balanced contexts. In the following, we introduce four

** Note that this language is not intended for programming at the user-level but rather a means to support the formal definition of its semantics and property analysis for aspects and components. The integration into mainstream programming languages as well as corresponding implementation support is discussed in [14].

operators (three pointcut operators and one advice operator) that extend our previous language and show how these operators benefit component evolution in the context of P2P systems.

Depth-dependent operators. In addition to restricting the depth of recursion as above, evolution of P2P distributed algorithms often aims at the optimization of the underlying traversal strategy through heuristics to perform a more superficial but faster search on nodes whose distance from the root node exceeds a certain threshold. Since VPAs faithfully allow to define the depth of nested terms, such heuristics can be directly expressed using a pointcut operator $D_m^{>k}$ that matches only calls to m that occur at a depth larger than k. For example, the following aspect caches queries at depth greater than 5 (where $\mu a. \ldots; a$ denotes recursion in VPA-based aspects):

$$\mu a.\ D^{>5}_{query_q} \rhd getCacheValue\ ;\ a$$

General sequencing operator. Aspect languages for protocols typically include a sequence operator ; on the pointcut level. Sequences of events in pointcuts may, however, match executions according to different semantics. Most frequently, the sequence $a; b$ matches an execution trace that contains the event a followed by a sequence of arbitrary events except b followed by the event b. Using this semantics — which has been pioneered by the approach of stateful aspects [4] and used in numerous others (*e.g.,* JAsCo [21] as well as our previous approach to VPA-based aspects [14] — the aspect

$$join\ ;\ \mu a.\ query_q \rhd saveContext\ ;\ a$$

waits for the current node to join a network and then repeatedly saves the local context (*e.g.,* on a backup server) if a query occurs.

This sequencing operator does not fit some common situations in evolution scenarios. Consider the two following cases:

- Components are extended by a backup operation that makes superfluous the (potentially costly) context saving but only if the backup is executed immediately after the query operation.
- General query operations must not be performed in certain cases, *e.g.,* if an erroneous situation occured or a certain neighbor has to be excluded from the search.

These scenarios are instances of two general problems: the sequence operator introduced above does not allow to define that occurrences of events have to immediately follow a particular event (even if the pointcut language includes a negation operator). Furthermore, it is tedious to exclude traces by excluding specific sets of individual events that are not allowed to occur.

In order to allow the concise definition of evolutions of the such kinds, *i.e.,* by arbitrary interleaving as well as through the (mandatory) absence of interleaving we propose a general sequencing operator $;_{\mathcal{I}}$ where $\mathcal{I}$ specifies the set of events that may be interleaved between the argument events ($\mathcal{I}$ may be $\emptyset$ to exclude any interleaving or the set of all events to allow arbitrary interleaving).

Permutation operator for well-balanced contexts. Permutation operators are frequently used for the construction of protocols that allow the arbitrary interleaving of sets of events. In the presence of well-balanced contexts, interleavings of calls as well as calls and corresponding returns are subject to restrictions that cannot be modeled simply using the standard permutation function that generates all permutations.

In P2P networks, for instance, a query on one node that triggers a query on a neighbor, *e.g.,* q_1 followed by q_2, must be followed by replies in the reverse order $\underline{r_2}$, $\underline{r_1}$. The permutation $q_1, q_2, \underline{r_1}, \underline{r_2}$ is not valid.

An advice operator to close open calling contexts. One major characteristics of a p2p network is the dynamism of peers. They can come and go unpredictably and thus it is common to have queries without corresponding replies. Subnetworks may be disconnected or messages lost. Error handling for those situations may involve the introduction of events that close a number of open recursive calls in order to skip the traversal of part of the underlying distributed network in which an error occured. Using VPA-based aspects such error handling strategies can be expressed using the advice-level operation $closeOpenCall_m$ that closes the open call to m: pointcuts matching on nested contexts can then be used to restrict the application of such advice to appropriate parts of the network. The following example illustrates the use of a closing operator to add a number of "fake" replies to queries when the query exceeds a given connection timeout (where $\square$ denotes the choice between alternatives):

$$\mu a.\ query_f; (\underline{reply_f} \ \square \ (connectionTimeOut \triangleright closeOpenCall_{query_f})) \ ; \ a$$

3.3 Syntax

Figure 4 presents the complete syntax of VPA-based aspects. (The operators introduced above (that extend the language of [14], are marked by boxes.)

A represents aspects which are defined by VPA-based expressions over basic aspects $P \triangleright Ad$ where P is a pointcut, and Ad is an advice action.

A pointcut P is constructed from terms T denoting method calls or returns or local operations (that may not influence the stack) as well as conjunctions and complements of terms. As discussed above, the sequencing operator $;_{\mathcal{I}}$ is the general sequencing operator where $\mathcal{I}$ specifies the type of events allowed to interleave between two other events. Furthermore, pointcuts allow regular expressions of events to be matched (remember that VPAs are strictly more expressive than finite-state automata). A pointcut can also be defined using depth-dependent operators (non-terminal D). Three different such operators are provided: D_M^{int} (nested execution to a specific depth), $D_M^{\leq int}$ (nested execution for with depths inferior to an integer value) and $D_M^{>int}$ (nested execution for depths greater than an integer value). Finally, pointcuts may be constructed using the two permutation operators $perms_{nested}(lst) and perms_{flat}(lst)$ that, respectively, allow to express concisely permutations concerning nested and flat pairs of calls and corresponding returns. These two constructors have been shown because its

$$
\begin{array}{lcl}
A & ::= & \mu a.A \\
 & | & \boxed{P \rhd Ad \;;_{\mathcal{I}}\; A} \\
 & | & \boxed{P \rhd Ad \;;_{\mathcal{I}}\; a} \\
 & | & A \,\square\, A \\
 & & \\
P & ::= & T \mid D \mid \boxed{Perms} \\
 & | & \boxed{\mathrm{P} \;;_{\mathcal{I}}\; \mathrm{P}} \mid P \| P \mid P\{\mathrm{int}\} \mid P+ \mid P* \\
T & ::= & Tl \mid !T \mid T \; and \; T \\
Tl & ::= & M \mid M_{Id} \mid \underline{M_{Id}} \\
D & ::= & D_M^{int} \mid D_M^{\leq int} \mid \boxed{D_M^{>int}} \\
Perms & ::= & \boxed{perms_{nested}(lst) \mid perms_{flat}(lst)} \\
 & & \\
Ad & ::= & send(M, Id) \mid \boxed{closeOpenCall(M, int)} \mid Ad \; ; \; Ad \\
 & & \\
M & ::= & \mathrm{const} \mid \mathrm{var} \quad // \text{ method names} \\
Id & ::= & \mathrm{const} \mid \mathrm{var} \quad // \text{ stack, component ids} \\
lst & ::= & \mathrm{list[M]} \quad //\text{list of methods}
\end{array}
$$

Fig. 4. Syntax of aspects over VPA-based protocols

frequent use as a building block for VPA-based protocols. Other permutation operators specific for well-balanced contexts can be defined but are less useful.

Advice Ad consists of finite sequences constructed from the operator for closing call contexts *closeOpenCall* and calls to services of named components. An application of the operator $closeOpenCall(m, n)$ inserts n calls to the return instruction corresponding to the call m.

3.4 Semantics

In this section, we informally present how the language extensions introduced in this paper can be integrated in the formal framework for the definition of small-step operational semantics for VPA-based aspects introduced in [14]. To this end, we first give a brief overview of that framework and then explain how to define the four new operators using this framework. Here in this paper, we will mainly discuss the semantics of the new extensions introduced in the previous subsection.

Overview of formal framework. The formal framework introduced in [14] defines the semantics of VPA-based aspects as a small-step semantics of the execution of woven program. Aspects (non-terminal A in Figure 4) are interpreted by repeatedly matching events of the execution of the base program with basic aspects $p \rhd a$. If the pointcut p matches, the advice a is executed and the

next basic aspect defined using the repetition, sequence and choice operators is determined.

Pointcut declarations are translated into a pointcut VPA (in the sense as defined by Alur and Madhusudan [2]). This pointcut VPA is then used during application of basic aspects to decide matching of, in particular, stack-manipulating calls and returns. Complex pointcuts that do not match individual execution events, such as regular expression pointcuts, correspond to paths in the pointcut VPA. As part of the operational semantics, program configuration containing the state of the pointcut VPA that evolves along with the base program execution.

Advice is inserted once the corresponding pointcut has matched by inserting the advice body — sequences of stack-manipulating return operations defined using the advice operator *closeOpenCall* or calls to component services — before, after or instead of the matched base execution event.

Definition of language extensions. In the remainder of this section we present how the extensions of the VPA-based language introduced in this paper can be defined using this formal framework.

Permutation operators. The two permutation constructors generate sequences of pairs of well-balanced pairs of calls and corresponding returns.

- The constructor $perms_{nested}(lst)$ generates nested pairs of call and return events. $perms_{nested}(query_a, query_b)$, *e.g.,* generates

  $$(query_a\ ;\ query_b\ ;\ \underline{reply_b}\ ;\ \underline{reply_a}) \square (query_b\ ;\ query_a\ ;\ \underline{reply_a}\ ;\ \underline{reply_b})$$

- The constructor $perms_{flat}(lst)$ generates flat sequences of pairs of call-return events. $perms_{flat}(query_a, query_b)$, *e.g.,* generates

  $$(query_a\ ;\ \underline{reply_a}\ ;\ query_b\ ;\ \underline{reply_b}) \square (query_b\ ;\ \underline{reply_b}\ ;\ query_a\ ;\ \underline{reply_a})$$

The two permutation constructors have straightforward formal definitions, for instance, in terms of suitable restrictions of the standard permutation function. The resulting permutation defining functions can then be used during pointcut matching to recognize traces of events corresponding to well-balanced permutations.

General sequence operator. The pointcut VPA is constructed bottom-up by starting from nodes representing basic pointcuts that match single execution events and link them through transitions according to the sequencing, repetition and choices used in a pointcut expression. That is, depending on the type of aspect composition operator used in the pointcut expression, we apply appropriate VPA operations such as adjunction of paths (for choice operator) and concatenation (for sequencing and repetition operators) to build the corresponding composed VPA.

The general sequence operator $;_{\mathcal{I}}$ can be formally defined as follows:

– Modify the matching semantics of the pointcut VPA, so that no interleaving of events is allowed between two consecutive matched events, *i.e.,* that transitions formalize compositions using the sequence operator $;_\emptyset$. More formally, if a pointcut p_1 matches the current execution event and if the VPA transition (p_1, p_2) has to be taken next, the next execution event must match p_2. (This is in contrast to the semantics used in [14, 4] that allow such interleaving.)
– Define the variants of the general sequence operator that allow interleaving, *i.e.,* $;_\mathcal{I}, i \neq \emptyset$, by inserting appropriate paths in the pointcut VPA.

Depth-dependent pointcuts. The pointcut VPA for D^n_m, which represents contexts starting with n open calls of m, can be easily constructed by n transitions representing m that may be interleaved with arbitrary deep well-balanced contexts involving m and $\underline{m}$. Then the pointcut VPA for the other two depth constructors $D^{\leq n}_m$, $D^{>n}_m$ are built from the VPA for D^n_m based on the following definitions:

$$D^{\leq n}_m := \|_{1 \leq i \leq n} D^i_m$$
$$D^{>n}_m := !D^{\leq n}_m$$

Hence, the VPA for $D^{\leq n}_m$ is result of the concatenation of individual VPAs with depth value from 1 to n and the VPA for $D^{>n}_m$ is the complement of that for $D^{\leq n}_m$, which is well-defined because VPA are closed under complement.

Call-closing advice. Advice application formally corresponds to the insertion of transitions corresponding to the advice body into base execution steps. In the small-step semantics, the program then produces the next base execution event and makes the pointcut VPA (and thus the aspect) evolve to the next state. The advice action *closeOpenCall(m, n)* is no different: its effect simply is the insertion of n return operations corresponding to the call m.

4 Preservation of compositional properties

In this section we address the problem whether compositional systems that are subjected to evolution by VPA-based aspects can be proven to preserve fundamental composition properties. Our main point is that, in contrast to general aspect languages such as AspectJ, VPA-based aspect programs are amenable to formal correctness proofs.

We use standard trace-based notions of compatibility and substitutability [23] in this paper. Two protocols are compatible if they do not give rise to any conflict during execution, *i.e.,* no unexpected message is received during collaboration of two components according to their respective protocols. As a simple example consider two nodes that both join the file sharing system and employ two following protocols:

– The first uses the protocol shown in Fig. 3 that allows queries to be aborted when an *abortRequest* is sent in state 2.
– The second uses a protocol that may issue abort requests at any time.

In this case the two protocols are not compatible because the protocol of the first node only allows abort requests to be handled in a specific state not all states.

Substitutability enables a protocol to be used instead of another one in arbitrary contexts. Substitutability of components is typically defined in terms of trace set inclusion: protocol p_1 is substitutable for p_2 if its trace set is a superset of the trace set generated by protocol p_2. For example, assume that p_1 is the protocol that includes nested calls to *query* and *reply* and p_2 is the protocol that consists of only non-nested calls to *query* and *reply*. Then p_1 is substitutable for p_2 since the trace set of p_1 covers that of p_2. Instead of in terms of trace sets only, more precise definitions of both notions can be defined by also taking into account failures over sequences of service requests, see [15].

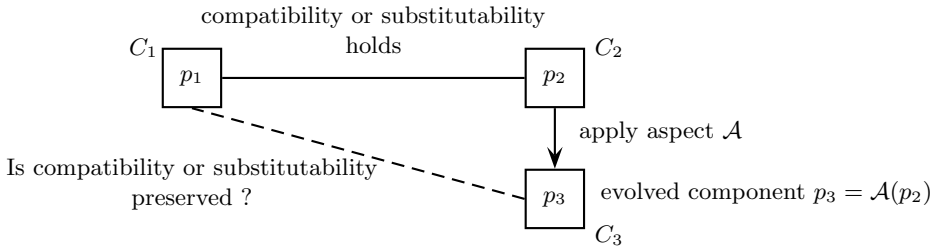


Fig. 5. Checking for preservation of compatibility/substitutability

Figure 5 illustrates the underlying model of component evolution and the compositional properties we consider. Starting from two protocols p_1, p_2 that constrain the interactions of two collaborating components C_1, C_2 a VPA-based aspect $\mathcal{A}$ is applied to p_2 yielding the protocol p_3 that defines the interactions of the component C_3 after evolution. As indicated in the figure we are interested in two fundamental correctness properties for components, compatibility and substitutability (see, *e.g.,* [15]).

Generally, *e.g.,* if turing-complete pointcut and advice languages are used for component evolution (as in AspectJ where arbitrary Java methods may be called in `if`-pointcuts and advice), such component properties cannot be proven formally. Furthermore, even in specific cases where a proof is possible, it can typically be performed only in terms of the woven program and not simply in terms of the aspects themselves. VPA-based aspects, however, support formal proofs of such properties because of their limited expressiveness and allow some important properties be proven simply by considering properties of the aspect language only. To this end we propose to exploit the "domain specific" characteristics of VPA-based aspects: proofs over nested contexts as well as regular structures can be performed directly in terms of corresponding features of our pointcut (indexed calls) and advice language (*closeOpenCall*).

Concretely, we demonstrate in the following three different types of proofs of property preservation that are supported by VPA-based aspects:

P1) Proofs that depend only on the properties of the aspect language, *i.e.,* that can be performed in terms of the evolution aspect $\mathcal{A}$ only.
P2) Proofs that can be performed in terms of $\mathcal{A}$ and properties of *classes* of protocols to which p_1 and p_2 belong.
P3) Proofs that require full knowledge of $\mathcal{A}$ and p_1–p_2.

In the following we will present three examples that illustrate the different proof types introduced above.

P1: supporting evolution of error handling. VPA-based aspects are unique (in particular compared to finite-state based approaches) in being able to handle a large class of traversals of distributed recursive algorithms, such as P2P algorithms. Frequently, error handling in such algorithms consists in terminating the exploration of some part of the network and search elsewhere. The action *closeOpenCall*(m, n) that we have introduced in the advice language directly supports such error strategies by allowing to close n nested calls of the method m.

We can exploit the precisely defined semantics and limited effect of the action *closeOpenCall* to prove some corresponding properties simply in terms of its definition. For example:

> If p_1,p_2 are protocols that recurse using m, p_2 is substitutable for p_1 and aspect $\mathcal{A}$ employs *closeOpenCall* to add returns of m at the end of the execution of protocol p_2, then the adapted protocol p_3 is substitutable for p_1.

Imagine that p_1 is the protocol consisting of recursive calls to *query* and p_2 is an extension of p_1 but explicitly requires one *reply* in its definition (*i.e.,* there can be more than one query but only one reply for the protocol to be complete). Since the trace set generated by p_2 is the superset of that of p_1, p_2 is substitutable for p_1 according to the definition of substitutability. Now assume that protocol p_2 evolves through an aspect that uses the *closeOpenCall* to add a number of *reply* to close all open *query* so that p_2 can work well with another protocol which requires an equal number of replies to the number of queries should be available. The new protocol p_3 resulting from that aspect-based evolution is still substitutable for p_1 since the trace set of p_1 is included in the trace set of p_3.

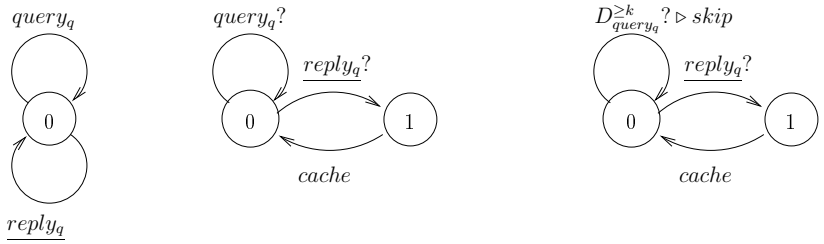
P2: proving compatibility for depth-cutting heuristics. Recursive distributed algorithms frequently do not unconditionally stop traversals at the top level, but typically do so only in specific contexts. A common example are heuristics that are formulated in terms of the traversal depth from the node where the search has been initiated. Since VPA-based aspect allow the explicit definition of aspects in terms of the nesting depth using the pointcut operator $D^{>k}_{m_c}$, corresponding compositional properties can be proven in terms of properties of this operator and classes of protocols to which it is applied. For example:

If
- p_1 belongs to the class of recursive protocols that repeatedly allows recursive remote calls and returns in m: $\mu a.m_c \,\square\, \underline{m_c}\ ;\ a$,
- p_2 belongs to the class of protocols that include a remote call to m,
- p_1, p_2 are protocol compatible and
- aspect $\mathcal{A}$ employs a depth-defining operator $D^{\geq k}_{m_c}$ applying over p_2

Then p_1 and $p_3 = \mathcal{A}(p_2)$ are also compatible.

This property holds because the aspect may only cut calls to m from traces of p_2: the resulting traces remain compatible with those admitted by p_1.

Hence, knowing the specific classes of protocols of p_1, p_2 and class of the aspect modifying them, one can use the above rule to conclude that compatibility is preserved. Let us consider a more intuitive example as illustrated in Figure 6. Fig. 6(a) shows protocol p_1 which provides basic implementation for recursive query including sequences of queries and replies. Fig. 6(b) illustrates the protocol p_2 that implements an advanced recursive query which allows to cache results. (Note that we use the '?' character to denote remote calls to services provided by another component). According to the definition of protocol compatibility, p_1 and p_2 are compatible. To adapt protocol p_2, an aspect is employed to dynamically skip the calls to *query* for depths greater than k, which is shown in Fig.6(c). We can prove that in this context, p_1 and p_3 are protocol compatible regardless of the change caused by the aspect.



(a) Protocol p_1: basic query

(b) Protocol p_2: extended query with cache

(c) Protocol p_3: adapted protocol of p_2 by aspect

Fig. 6. Aspect-based evolution implemented by depth-defining operator

P3: proving substitutability in terms of p_1,p_2 and $\mathcal{A}$. In this example, the classes of protocols and aspect are not sufficient to prove the preservation of compatibility and we need more information on the real values of p_1,p_2 to prove component properties.

Let us reconsider protocols p_1,p_2 as in the first example *i.e.,* p_1,p_2 involve recursive calls to *query* and p_2 is substitutable for p_1. Assume that now we would like to adapt protocol p_2 in order to cut the depth of queries to k using an aspect

with a depth-defining operator $D_{m_c}^{>k}$. In this case the resulting protocol p_3 is in general not substitutable for p_1, since p_1 may admit calls of depth deeper than k. By an analysis of p_1, we may find that the depth limit of p_1 is q and $q \leq k$: we can then prove that p_3 is actually substitutable for p_1.

5 Related work

There is few related work on aspect-based evolution for component-based systems that considers the preservation of correctness properties for those systems after being changed by aspects. As to the best of our knowledge, this proposal is the first exploiting formal methods to investigate the preservation of compositional properties such as compatibility and substitutability for component-based systems that are subject to evolution by protocol-modifying aspects. However, our work still shares common interests with a large body of work covering aspects, components, and applications of formal methods on analysis and verification.

There are some approaches which consider aspect languages that support protocols, most notably [1, 5, 22]. Approaches [1, 5] feature regular aspect languages and a framework for static analysis of interaction properties. The language introduced by Walker and Viggers[22], one of the very few approaches providing non-regular (but not turing-complete) pointcut languages, proposes tracecuts which provide a context-free pointcut language. However, all of the above approaches do not use the language for an integration of aspects and components or explore the problem of property-preserving for systems that have protocols being modified by aspects. Farías [9] has proposed a regular aspect language for components that admits advice modifying the static structure of protocols and considered proof techniques for the resulting finite-state based aspects.

There exist a large number of proposals that aim at applying AOP over component-based systems, *e.g.,* [10, 20, 16]. However, the aspect languages in those approaches do not provide explicit support for component protocols. Some of these approaches consider component compatibility, however, in a limited sense to our work: aspects are usually employed in such work to transparently introduce adaptation to components and thus preserve component compatibility. Our approach, in contrast, focuses on preserving protocol compatibility even if aspects have visible effects on interaction protocols.

Few work on evolution of component protocols seems relevant to our work. Braccialia et al. [3] present a formal methodology for automatically adapting components with mismatching interacting behaviors *i.e.,* conflicts at the protocol level. Protocols considered there are expressed by using a subset of $\mu-$calculus. They do not consider how component properties can be proved in terms of proof methods that exploit properties of modification operators. Ryan and Wolf [18] investigate how applications can accommodate protocol evolution. However, this approach concerns mainly syntactic changes on protocols.

Another category of related work is the application of formal methods to analyse aspect systems, such as [11, 12]. Our approach differs from those approaches

in that we exploit the protocol-based specificities of our aspect language to prove composition properties of software components.

6 Conclusion

In this paper we have motivated the use of aspects to define the evolution of components with explicit interaction protocols. We have motivated and introduced four extensions to our previously defined VPA-based aspect language that are useful in the context of component evolution for distributed applications: a depth-dependent operator, a general sequencing operator, two permutation operator for well-balanced contexts, and an advice operator to close open calling contexts. The first three extensions improves the expressiveness of the pointcut language and make it possible to express common evolution aspects more precisely. The last extension enables aspects to modify interaction protocols in a limited manner, *e.g.,* to support error correction strategies.

Furthermore, we have addressed the problem of preserving compositional properties such as compatibility and substitutability for components that are subject to aspect-based evolution. The main innovation in our approach to handle this problem consists in the exploitation of the aspect language features to reason about properties of components modified by aspects. Concretely, we have shown that our VPA-based aspect language of limited expressiveness admit formal proofs of fundamental compositional properties in the presence of aspect-based evolutions directly in terms of the aspect languages.

With respect to future work, we plan to advance in two directions: language design and proof techniques for property preservation. We strive at the definition of a larger set of VPA-based pointcut constructors for distributed components; furthermore a more powerful advice language for protocol modifications should be included. Moreover, property analysis should take into account modifications made by aspect advice.

References

1. 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.
2. 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.
3. Andrea Braccialia, Antonio Brogi, and Carlos Canal. A formal approach to component adaptation. *Journal of Systems and Software*, 2005.
4. R. Douence, P. Fradet, and M. Südholt. A framework for the detection and resolution of aspect interactions. In *Proc. of GPCE'02*, LNCS 2487, pages 173–188. Springer Verlag, October 2002.

5. 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.
6. 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.
7. Andrés Farías. *Un modèle de composants avec des protocoles explicites*. PhD thesis, École des Mines de Nantes/Université de Nantes, December 2003.
8. 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.
9. 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.
10. Steffen Göbel, Chrstoph 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.
11. Shmuel Katz and Marcelo Sihman. Aspect validation using model checking. In *Verification: Theory and Practice*, pages 373–394, 2003.
12. Shriram Krishnamurthi and Kathi Fisler. Foundations of incremental aspect model-checking. *ACM Trans. Softw. Eng. Methodol.*, 16(2):7, 2007.
13. Qin Lv, Pei Cao, Edith Cohen, Kai Li, and Scott Shenker. Search and replication in unstructured peer-to-peer networks. In *ICS*, pages 84–95, 2002.
14. Dong Ha Nguyen and Mario Südholt. VPA-based aspects: better support for AOP over protocols. In *4th IEEE International Conference on Software Engineering and Formal Methods (SEFM'06)*. IEEE Press, September 2006.
15. Oscar Nierstrasz. Regular types for active objects. In O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Composition*, chapter 4, pages 99–121. Prentice Hall, 1995.
16. Nicolas Pessemier, Lionel Seinturier, Thierry Coupaye, and Laurence Duchien. A model for developing component-based and aspect-oriented systems. In *Proceedings of the 5th International Symposium on Software Composition (SC06)*, volume 4089 of *Lecture Notes in Computer Science*, page 259273, Vienna, Austria, mar 2006. Springer-Verlag.
17. Frantisek Plasil and Stanislav Visnovsky. Behavior protocols for software components. *Transactions on Software Engineering*, 28(9), January 2002.
18. Nathan D. Ryan and Alexander L. Wolf. Using event-based translation to support dynamic protocol evolution. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 408–417, Washington, DC, USA, 2004. IEEE Computer Society.
19. 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.
20. 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.
21. W. Vanderperren, D. Suvee, M. A. Cibran, and B. De Fraine. Stateful aspects in JAsCo. In *Proc. of SC'05*, LNCS 3628. Springer Verlag, April 2005.

22. 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.
23. 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.