

cHALO, stateful aspects in C

Bram Adams
GH-SEL, INTEC, Ghent University
Sint-Pietersnieuwstraat 41
B-9000 Ghent, Belgium
bram.adams@ugent.be

Charlotte Herzeel and Kris Gybels
PROG, Vrije Universiteit Brussel
Pleinlaan 2
Brussels, Belgium
{charlotte.herzeel,kris.gybels}@vub.ac.be

ABSTRACT

History-based pointcut languages are a very expressive and powerful means to obtain robust pointcuts. To implement them in an efficient way, people have proposed various optimisations and program history retention strategies, especially for Java-based aspect languages. In this paper, we focus on history-based pointcut languages for C, which has no provisions for weak references or automatic garbage collection. Because C programmers want explicit control over pointcut behaviour and memory footprint, we claim that a limited set of fine-grained temporal pointcut primitives with well-known memory behaviour strikes a good balance between flexibility and memory consumption. A working prototype (cHALO) has been designed and implemented, based on the HALO pointcut language for Lisp.

1. INTRODUCTION

To decouple aspects from the base code, robust pointcuts are indispensable. This desire has fostered development of expressive aspect languages. Early on [11], time has been identified as an important foundation for powerful pointcut languages. More specifically, pointcuts can be written as patterns over the program execution history. This program trace corresponds to a sequence of AspectJ-like join points. In general, the various pointcut designators making up a complete temporal pointcut¹ are composed via temporal operators or some other means (e.g. regular expressions) to detect certain patterns of events. Another interpretation is that the particular join points a pointcut is interested in may change depending on earlier events in the base code (stateful aspects [8]). This lends itself naturally for modeling protocol-like concerns [1, 9, 14] or complex behavioural patterns [3].

Despite its promises, adoption of history-based pointcuts has been slow to catch up [4], primarily because of efficiency issues. There are roughly two areas of concern: memory space and execution time. Intuitively, the idea of matching join points based on program trace elements requires that these traces and available context are stored somewhere before they can contribute to a pointcut

¹We call them “sub-pointcuts” from now on.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...\$5.00.

match². Worse, some state may need to be retained indefinitely, as the right future event leading to a pointcut match never seems to occur. In the meantime, context values associated with past events may have gone out of scope, rendering them invalid for future (partial) matches. As for execution time overhead, the process of deciding whether an event satisfies a given pointcut based on past program events is in general not statically decidable. A run-time component is needed to make the final residual checks for a match, introducing overhead.

For both problem areas, solutions and workarounds have been introduced. In this paper, we focus especially on program history retention strategies, i.e. policies which limit the number of needed trace data elements in memory and/or the period during which they are stored. Our claim is that current solutions hide too much complexity from the programmer, without the possibility to get in charge if desired. This clashes with the spirit of C, similar to the way C programmers despise garbage collection for taking away too much power/freedom from them. Additionally, existing program history retention techniques almost all exploit garbage collection facilities of the underlying base language. Systems or infrastructure software, traditionally implemented in C or C++, cannot benefit from garbage collection. Hence, history-based pointcuts may cause serious memory issues in these systems. *Lack of memory control and of explicit retention strategies in the absence of garbage collection, severely limit adoption of history-based pointcut languages in C/C++ applications.*

Although we do not focus directly on execution time overhead (many analyses [6, 7] and optimisations [4, 5] have been proposed for this), the latter is indirectly influenced by memory behaviour. Just as for memory usage, C programmers want to avoid redundant behaviour too. History-based aspect languages with too coarse-grained temporal operators and sub-pointcut constructs do not satisfy these needs, as they do not allow sufficiently fine-grained join point selection. Typically, more program history is stored for future matches, but undocumented links between operators and their memory consumption make this opaque for the developer. Hence, *history-based pointcut languages in C need fine-grained temporal operators with clear guarantees on memory behaviour.*

To deal with history retention and fine-grained operators, we propose to apply ideas introduced by HALO [14], an expressive history-based pointcut language for Lisp. It features a limited set of logic-based pointcut primitives with well-known memory space and cleanup characteristics, and a run-time weaver implemented in terms of a slightly customised Rete-engine [12]. *HALO’s temporal operators and their history retention strategies are clearly mapped onto join nodes of the Rete-network*, hence improving the (memory) behaviour of operators or adding new ones is straightforward.

²This kind of incomplete matches are called “partial matches”.

```

1 void f(char* s, int d);
2 void g(char* s);
3 BOOL shorter(char* s, int d);

```

Figure 1: Interface of running example used in this paper.

```

1 seq(call(void f(char*, int)) && args(s, d)
2         && if(shorter(s, d)));
3     call(void g(char*)) && args(s2)
4         && if(s==s2) then ...

```

Figure 2: Arachne pointcut with dots instead of concrete around-advice.

We have translated these ideas to *Aspicere* [2], an aspect language for C with a static weaver. The resulting system, named “cHALO”, will be discussed. Benchmarking cHALO’s effective memory behaviour is considered future work.

Our position statement is that:

- History-based pointcut languages for C/C++ need to provide the programmer with explicit control over behaviour and memory footprint of temporal pointcuts.
- The clear mapping between temporal operators and join nodes of the Rete-network, makes HALO an ideal choice for providing the aspect developer with more control.

Section 2 discusses the two most related history-based pointcut languages, *Arachne* [9] and *tracematches* [3]. Then, the required HALO features for our explanation is presented in Section 3. We use these to rephrase the basic temporal operators of *Arachne* and *tracematches* in terms of HALO’s primitive predicates (Section 4). Section 5 then converts HALO’s key ideas in the context of a statically woven aspect language for C, and the shortcomings of the resulting system are discussed in Section 6. Finally, Section 7 summarises this paper’s contributions.

2. RELATED WORK

There are dozens of history-based pointcut languages. Because of space restrictions and to keep our discussion focused, we only consider three of them in this paper: *Arachne*, *tracematches* and *HALO*. This section focuses on the first two, while *HALO* is discussed in the next section. To illustrate the claims of the introduction, we use the simple example shown in Figure 1. It consists of two functions, *f* and *g*, and a boolean function which checks whether the length of a string (*s*) is shorter than a given integer (*d*). We model a history-based pointcut which looks for (possibly non-consecutive) invocations of *f* and *g*. These invocations should have the same string as their first argument. Furthermore, we are only interested in invocations of *f* for which *shorter* returns *TRUE*.

2.1 Event-based AOP and *Arachne*

Event-based AOP (EAOP) [11, 10] is based on the concept of execution monitors. Join points are reified as events in the execution of a program and pointcuts describe patterns of events. Douence et al. [11] have proposed a formally defined, domain-specific aspect language with operators for modeling event sequences, parallel processing, filtering, etc. A Java prototype has been developed in which the monitor is synchronously called by the base program. The specific instrumentation points where the monitor is invoked are automatically generated from the pointcut. The distinction between a run-time component (monitor) and event broadcasting from within the base code (instrumentation) is still widely used.

```

1 tracematch(String s, int d){
2   sym f around: call(void f(..)) && args(s, d)
3                 && if(shorter(s, d));
4   sym g around: call(void g(..)) && args(s);
5
6   f g{
7     ...
8   }
9 }

```

Figure 3: Tracematch equivalent to the *Arachne* pointcut in Figure 2.

Another implementation of EAOP has been proposed by Åberg et al. [1]. They use rewriting rules, expressed in temporal logic, to statically select join point shadows. This static transformation exploits the intra-procedural control flow graph of a program to avoid run-time overhead.

The most advanced incarnation of EAOP is *Arachne* [9], an expressive aspect language for C sporting a Prolog-influenced pointcut language and a run-time weaver. Only the sequencing construct of EAOP has been retained, but it forms the backbone of *Arachne*. Figure 2 shows a pointcut that matches a sequence of calls to *f* (lines 1–2) and *g* (lines 3–4) which fulfil two conditions: their first argument should be equal and they should pass the *shorter* boolean check. *Arachne* allows to associate advice to both sub-pointcuts of the sequence, not only to the sequence as a whole. Hence, the advice represented by “...” is actually around-advice on the call to *g*. This clearly gives aspect developers more control, but it can be misleading in the sense that advice is executed for any partial match, regardless whether or not it will eventually yield a complete match.

For memory management, *Arachne* associates a linked list with each sub-pointcut of a sequence (except the last one). Each list node stores the contents of a partial match, with values for all context variables which occur in the whole pointcut. At any moment, there can be multiple partial matches associated with a given sub-pointcut. Conceptually, when a new trace element matches a sub-pointcut, every partial match of the previous sub-pointcut is examined to check whether bound variables match. If they do, the node’s state can be updated and moved to the current sub-pointcut’s list. The associated advice is executed. The partial match’s node is freed (returned to a pool) if the last sub-pointcut has matched. If some partial match never yields a complete match, the accompanying set of bindings remains indefinitely in a linked list. This is actually inevitable, but we come back to this in our discussion.

2.2 Tracematches

Tracematches [3] consist of a set of events (symbols) which are considered interesting, a regular event expression in terms of those symbols and an advice body which is activated once the pattern is satisfied. Symbols correspond to a combination of an AspectJ advice kind and a primitive pointcut. A tracematch model of the *Arachne* pointcut of Figure 2 is shown in Figure 3³. Two symbols are declared (*f* and *g*). Prolog-like unification of (and backtracking over) the string arguments is enforced by reusing the same free variable name (*args(s)*). This forms the biggest difference between *tracematches* and prior history-based pointcut languages, and has had a big impact on memory behaviour (more on this later). Line 6 contains a simple regular expression in terms of the two symbols. It expresses the same sequence as the *Arachne* implementation does.

³Tracematches are expressed in Java, whereas *Arachne* and *Aspicere* are based on C. For this example, only the temporal pattern matters.

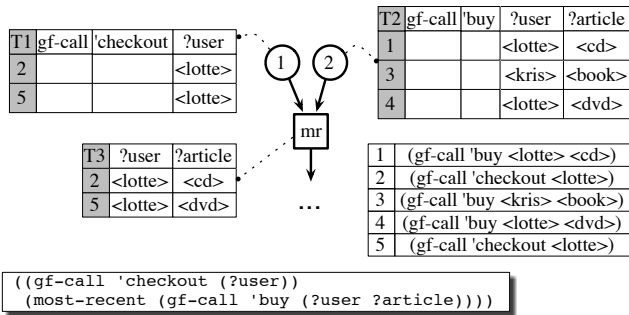


Figure 4: Example time-extended Rete network from HALO.

Contrary to Arachne, tracematch advice applies only to the last join point matched by the pattern, not on the whole join point sequence.

As introduced by EAOP, the tracematch weaver conceptually distinguishes between base code instrumentation and run-time support for deciding whether or not there is a match. Tracematches are translated into multiple AspectJ advices. First, some of these advise the appropriate instrumentation join points with event signaling logic. A second kind of advice contains specialised code to set up and run the run-time component. The third kind of advice contains the actual tracematch advice code. Many optimisation techniques have been proposed to optimise either the instrumentation or run-time side of the woven system [4, 5, 6, 7].

Tracematches use a specialised, deterministic finite automaton to keep track of the matching process. The automaton is automatically generated from and specialised to the tracematch. As automata are not designed to hold memory (for partial matches), some advanced concepts have been added. Every state is labeled by a number of disjunctive constraints to record all partial matches for a given state in the automaton. Upon arrival of new events, the constraints are incrementally updated. Conceptually, partial matches are moved to the next state until they disappear from the automaton. It is not clear whether this automaton-based approach easily allows extending it with new temporal operators or history retention strategies.

The biggest problem in the tracematch implementation is the need to avoid memory leaks [3, 5] in the presence of free variables. Apart from HALO [14], tracematches is one of the only approaches worried about this. The problem is that bindings inside a partial match suddenly could refer to garbage-collected data. To resolve this, tracematches sometimes use weak references to hold context variables. If a weak reference is garbage collected, the partial matches referring to it are also cleaned up. Still, getting this memory behaviour correct proved to be very hard [3, 5]. As is the case with Arachne, leaks are inevitable if non-garbage collected partial matches do never contribute to a complete match.

3. HALO

HALO [14] is a history-based pointcut language for Lisp. It provides a limited number of temporal operators, all of which relate an outer sub-pointcut to an inner sub-pointcut (two in case of `since`):

a most-recent b A new match for outer sub-pointcut `a` only matches with the most recent partial match of inner sub-pointcut `b` with which it can unify bound context variables.

a cflow b Similar to `most-recent`, but the join point satisfying `a` should lie inside the control flow of a join point which satisfies `b` (i.e. AspectJ’s `cflow`).

a since b c A new partial match for outer sub-pointcut `a` matches

with the partial matches of inner sub-pointcut `c` with which bound context variables can be unified and which have occurred later in time than the most recent partial match satisfying inner sub-pointcut `b`.

a all-past b A new partial match for outer sub-pointcut `a` matches with all partial matches of inner sub-pointcut `b` with which bound context variables can be unified.

We have listed these operators based on their memory footprint, from low to high. These memory requirements are transparently linked to the implementation of the HALO weaver [14]. This is a dynamic weaver based on a Rete engine [12] which has been extended with time stamps and extra logic corresponding to the temporal operators. The Rete algorithm is a forward chaining technique specialised in checking whether a sequence of asserted logic facts satisfies a logic formula (in the case of HALO: a logic pointcut matches a join point).

The Rete algorithm is based on a network of nodes, node connections and memory tables attached to the nodes, as shown on Figure 4. Round nodes are “filter nodes”, as they filter newly asserted facts based on specified values for logic variables⁴. The filter nodes correspond to the conditions from the logic formula. In HALO, they correspond to sub-pointcuts. The rectangular node is a “join node”, which tries to unify a new partial match which has entered via the node’s left or right input node with the other input’s partial matches. If there is a match, the node memorises it and sends the new match to its output. Partial matches are stored within memory tables attached to filter and join nodes.

By default, join nodes perform a logical “and” on the partial matches in their input nodes’ memory tables. In HALO, each temporal operator is associated to a custom join node (multiple in the case of `since`). The one on Figure 4 e.g. represents the `most-recent` operator in the HALO pointcut at the bottom. Instead of just performing a logical “and”, the time stamps which have been added by HALO to memory tables (gray columns) are taken into account. Upon a new partial match in its left input node, the `mr` join node combines this entry with the most recent matching entry in the memory table of the right input node. This means that both entries need to have the same values for common variables (like a normal “and” requires) *and* that the time stamp of the right partial match should be lower than the time stamp of the left entry. For this reason, if the `mr` gets a new partial match from its right input without a match coming from the left input, there will be no match.

To illustrate HALO’s weaver implementation, Figure 4 shows how the network has processed the program trace on the lower right. The asserted fact of time stamp 1 is only inserted and memorised by the right filter node, because it is a call to `buy` with two arguments. The join node does nothing, because it cannot match with the empty memory table of its left input. The fact at time 2, however, does trigger a full match, as it matches the left filter node and can be unified with an older fact memorised by the right filter node. The resulting match is stored inside the join node’s memory table and is also sent to the output of the network to signal a full match to the weaver. By the time the fifth fact has been asserted, a pure Rete join node would match it with the partial matches of time stamps 1 and 4. However, the `mr` node is only interested in the most recent partial match on the right which matches with its left input, and hence only one full match is made and stored.

Until now, we have only focused on the direct mapping between temporal pointcuts and Rete network. Contrary to approaches like Alpha [15], the naive memory requirements of Figure 4 can be drastically optimised [14]. There are various history retention strategies

⁴Variable names start with a ‘?’.

```

1 (gf-call 'f <aa> <1>) 4 (gf-call 'g <bb>)
2 (gf-call 'f <aa> <3>) 5 (gf-call 'g <aa>)
3 (gf-call 'f <aa> <4>) 6 (gf-call 'g <aa>)

```

Figure 5: Sample program trace for the system in Figure 1.

```

1 ((gf-call 'g ?s)
2 (since (most-recent (gf-call 'g ?s))
3 (all-past (gf-call 'f ?s ?d)
4 (if 'shorter ?s ?d))))

```

Figure 6: HALO pointcut which corresponds to the Arachne pointcut in Figure 2 and the tracematch in Figure 3.

possible, based on time stamps and on the known behaviour of the temporal operators. Because the `mr` node is not connected with any other join node and just sends its partial matches to the output of the network, it does not need to store its partial matches, and hence does not need a memory table. A similar remark holds for the left filter node, as a newly asserted fact is not used anymore once it has been sent to the join node. Finally, the semantics of the `most-recent` operator suggests that duplicate partial matches stored within the filter node’s right input can be removed, as only the most recent one is interesting. The interpretation of “duplicate” is more general than usual, as in the context of the `most-recent` operator it boils down to “having identical values for variables which are common between the inner and outer sub-pointcut”. Other variables are not used during matching of the left and right partial matches. Hence, in the example, fact 1 becomes obsolete once fact 4 is stored in the right memory table, even though they have different values for the `?article` variable. This means that at time 5, the network in Figure 4 only requires memory space for two facts in the right input’s memory table, nothing more.

With these concepts in mind, we now revisit Arachne and tracematches to express their behaviour and memory requirements in terms of HALO’s temporal operators. This enables better understanding of the problems with the memory footprint.

4. RELATED WORK REVISITED

To better understand the semantics and memory requirements of Arachne and tracematches, we try to rephrase the example Arachne and tracematch implementations of Figures 2 and 3 in terms of HALO. As a side-effect, this exercise gives an indication on the fine-grainedness of these three pointcut languages with reference to each other, and on the link between temporal operators and memory management. We use the sample program trace in Figure 5 for the running example of Figure 1. Despite its simplicity, it suffices to convey our main message.

For the running example, the Arachne and tracematch implementations conceptually yield the same program output and partial match memory consumption. More precisely, the first event is ignored, as the integer (1) is smaller than the length of `<aa>` (2). The next two invocations do match the first sub-pointcut of the sequence and tracematch. Both approaches record the bindings (`?s -> <aa> ?d -> <3>`) and (`?s -> <aa> ?d -> <4>`). The next three events try to extend the partial matches into a complete match. The fact on line 4 does not succeed, since `<aa>` differs from `bb`. The next one triggers two complete matches, one per partial match. Hence, the advice is executed twice, i.e. for (`?s -> <aa> ?d -> <3>`) and for (`?s -> <aa> ?d -> <4>`). Arachne now removes the two partial matches from the linked list associated with the sequence’s first sub-pointcut. Tracematches do the same by manipulating the constraints associated

with the automaton’s first state. This is important, as it means that the last event in the trace (line 6) does not trigger any advice. There are simply no partial matches left anymore.

So far, we have considered the program output and we have also observed that the operator behaviour affects the retention of partial matches. Arachne and tracematches by default retain every matching event as a partial match for a given sub-pointcut until it gives rise to a longer partial match. This partial match moves on to the next linked list or state, or it disappears on a complete match. Note that tracematches can also remove partial matches. Negative symbols (left out from a pattern) cause constraints to disappear from an automaton state. Analyses and optimisations also indirectly influence the bindings. They are especially focused on shifting as much decision logic as possible to compile-time, or to accelerate access to partial matches [4, 5, 6, 7]. Handling retention of program history is merely a side-effect, not their primary goal. As these analyses are not enforced but optional, programmers do not have control over the retention policy. This means that, in our example, all corresponding invocations of `f` need to be retained between two consecutive full matches for a given string argument. Assuming that these invocations are more frequent than calls to `g`, a lot of space can be required.

We can express the execution and memory behaviour of Arachne and tracematches using HALO concepts. Figure 6 shows the corresponding pointcut. The previous full match for a given string `?s` is given by `most-recent (gf-call 'g ?s)`, while the pointcut expression on lines 3–4 selects all past calls to `f` which satisfies the `shorter` test. Taken together, lines 2–4 collect all calls to `f` for a given string `?s` since the previous full match of the whole pointcut. This is the set of partial matches with which each new call to `g` can be combined to form a complete match. This corresponds to the behaviour and memory requirements of the Arachne and tracematch pointcuts. The memory footprint consists of one entry for recording the context of the last call to `g` for any given string `?s`, as well as facts for keeping the context of all calls to `f` since the last corresponding `g` invocation.

From this, we can make a couple of observations. First, the basic history-based pointcut primitives in approaches like Arachne and tracematches are actually coarser-grained w.r.t. the temporal primitives HALO provides. The reason is that both want to make sure that any event which gives rise to an initial partial match eventually will get a chance to form a complete match. No partial match is considered redundant or a duplicate, except when there are negated sub-pointcuts, a new sub-pointcut is matched or when a binding is garbage-collected. This behaviour is sometimes desired, but in many use cases it suffices to act just once on a series of more or less identical events. If a USB device is connected, it may broadcast several “I am new” events, but the operating system’s I/O component just needs to know there was at least one such an event. Network protocols, timer alarms, scheduling requests, etc. could be implemented using the simpler, more fine-grained temporal operators of e.g. HALO to reduce unnecessary execution and memory overhead. The coarse-grained pointcuts can still be composed from these fine-grained operators, but at the cost of additional execution time and memory consumption.

A second observation is a consequence of the above coarse-grained language semantics. As illustrated by the extensive coverage of memory leaks by the tracematch team [3, 5], the memory requirements of history-based pointcut implementations are rather opaque and implicit. This conflicts with the spirit of C and C++, as explicit, intuitive control over memory is one of the hallmarks of these languages. It is not immediately clear how to map the memory retention policy of a temporal operator to one specific concept of the run-

```

1 ((gf-call 'g ?s)
2 (most-recent (gf-call 'f ?s ?d)
3 (if 'shorter ?s ?d)))

```

Figure 7: Finer-grained sequence pointcut in HALO.

```

1 void fg_sequence(char* S, int D)
2     around [around F, around G]:
3     invocation(F, "f") && args(F, [S, D])
4         && if(shorter, S, D) =>
5     invocation(G, "g") && args(G, [S]) {
6     ...
7 }

```

Figure 8: Aspicere pointcut corresponding to the HALO pointcut of Figure 7.

time components. In HALO, by design an operator corresponds to one specific join node (or two) of the Rete network with guarantees on memory retention of the associated partial matches. This also means that adding a new operator boils down to introducing a new node type. As a result, the aspect developer has more control. This gives systems software developers the possibility to tailor temporal pointcuts to the application at hand, and to give them the opportunity to make well-founded assessments of memory requirements and execution speed.

To validate these observations, we have translated some of HALO’s key features to an existing aspect language for C, Aspicere [2], and have modified Aspicere’s weaver for it. We present the resulting system, named “cHALO”, in the following section.

5. CHALO

Figure 7 shows a variant of the Arachne and tracematch pointcuts from Section 2) similar to Figure 4. A call to `g` can trigger at most one complete pointcut match, because there can only be one `f` fact in memory with the same values for common bindings (see Section 3). The other facts have been identified as duplicates (based on `?s`) and subsequently have been removed. In many cases, this pointcut’s behaviour suffices instead of the coarser-grained semantics of Figures 2 and 3. Furthermore, for the example of Figure 7, at most one fact is stored for the calls to `f` in the trace of Figure 5, while none of the calls to `g` warrants storage. Note that the most recent partial match satisfying the inner pointcut is not removed upon a complete match [14], contrary to Arachne [9] and tracematches [3]. We come back to this later.

Figure 8 shows the corresponding pointcut (and advice) in cHALO, i.e. the history-based extension of the Aspicere aspect language [2]. Aspicere has a Prolog-based pointcut language, and advice corresponds to regular C code in which type parameters and typed context can be used. Lines 1–2 tell us that `fg_sequence` is around-advice which matches on a sequence of two symbols, `F` and `G`. It provides two context variables to the advice, i.e. the string (`char*`) and the integer arguments. Lines 3–5 show that `F` and `G` correspond to invocations (calls) to the `f` and `g` procedures mentioned in Figure 1. Their arguments are captured and the `F` symbols are filtered by the `shorter` boolean function. The `=>` arrow identifies the semantics of the sequence, in this case a `most-recent` operator.

HALO has a dynamic weaver which is centered around a slightly customised Rete [12] engine, whereas Aspicere sports a link-time weaver based on a Prolog engine. These two approaches are not at odds with each other. As we have seen in Section 2.1, most temporal aspect languages [11, 4] conceptually are implemented

via a combination of base code instrumentation and run-time decision logic. The former makes sure that only interesting events are signaled, while the latter decides at run-time whether or not the recorded join points match a pointcut. As identified by Avgustinov et al. [4], both components can be optimised independently to reduce memory footprint and execution time. Using static analysis, one can filter out join point shadows which can never lead to a match, specialise the run-time logic to the pointcuts in use or even eliminate run-time checks altogether.

Hence, for cHALO we have decided to apply Aspicere’s Prolog-based link-time weaver to pinpoint suitable instrumentation join points, while a run-time Rete-engine is linked with the woven application to perform the run-time checks. Instrumentation then amounts to asserting facts to the engine, and checking whether a rule has been triggered, in which case advice should be executed. Static analysis can be used to limit the number of join point shadows where facts need to be asserted. Run-time optimisation can be achieved by automatic garbage collection of facts based on the temporal operators’ semantics.

With this infrastructure in place, we are also able to experiment with new, finer-grained temporal operators. More precisely, operators can be conceived whose semantics follows directly from the desired memory footprint. In practice, this amounts to adding new kinds of Rete join nodes, with specific matching and memory cleanup behaviour. As an example, we have added and implemented a `very-most-recent` operator as an extension to the `most-recent` operator which discards the partial match of the inner sub-pointcut on a match with a new event. This avoids future matches with such a partial match. In the advice shown on Figure 8, one just needs to replace `=>` by `~>` for this. This operator gives the aspect developer more explicit control over semantics and memory footprint in case this is needed. Other operators like `all-past` and `since` have not been added yet to cHALO, but in principle the same approach can be followed.

6. DISCUSSION

Explicit program history retention strategies can only stabilise the memory footprint of history-based pointcuts if the latter are thought out well. Suppose that in the example trace of Figure 5, invocations to `f` with identical string arguments are very rare. This would mean that the garbage collection optimisation does not kick in, and almost all calls to `f` are memorised. This could get worse when calls to `g` would also be infrequent, or not contain matching string arguments. In that case, even the `very-most-recent` operator would not be able to reduce the memory footprint. The same problems exist in Arachne and, to a lesser extent (because of weak references), in tracematches.

In addition to memory starvation, there is another danger. If a very old partial match suddenly leads to a complete pointcut match, chances are high that its context variables are not valid anymore. They could refer to local variables which have long gone out of scope, or to pointers which have become dangling by now. Tracematches can easily deal with this via weak Java references, but in C or C++ no such mechanism exists, simply because there is no (automatic) garbage collection. This is the biggest obstacle history-based pointcuts are facing in the context of C and C++ systems. Providing more explicit control on memory requirements of temporal pointcuts, as proposed in this paper, is one initial step towards a solution. A more complete approach could e.g. advise any function with instrumented shadows to invalidate all partial matches which bind local variables, or use some kind of smart pointer to refer to bound context. In any case, more research is required to solve these problems.

In our examples, we have used a boolean function to check a condition on captured context. This is more tricky than it appears on first sight, as it is closely related to HALO's concept of "future variables" [13, 14]. Approaches like Alpha [15] check for matches using a backward chaining logic language. As such, the current values of context variables are used to match past facts. HALO's forward chaining weaver semantics works the other way around. Facts have been asserted in the past with the then current value of context bindings. This value may differ from the one in use when the complete pointcut eventually matches. This especially is relevant in the case of pointers, where the value pointed at may have changed in the meantime.

Conceptually, a partial match's context variables should retain their original value for as long as they are stored in memory, although several gradations of this behaviour could be devised [13]. In all cases, a flavour of deep copying is required to make future variables work in C or C++. This is not easy, as making a deep copy of a generic `void` pointer is not possible. Ideally, the weaver should generate specialised deep copy algorithms for each declared context type, but this is not straightforward. Currently, cHALO takes shallow copies of context variables, i.e. only the address pointed at by pointers is copied, not the values they refer to.

Finally, we have no concrete figures yet on execution time and memory usage in real systems. We need to find a suitable case study on which we can compare e.g. Arachne and cHALO. Optimisations in cHALO's execution speed may be required, but this falls outside the scope of this paper.

7. CONCLUSION

In this paper, we have claimed that current history-based pointcut languages do not blend well with systems software, because they do not provide the programmer with sufficient control over memory usage. Existing program history retention strategies have been designed explicitly for languages with garbage collection. In addition, the connection between a temporal operator's behaviour and its memory footprint is mostly unknown, and is considered an implementation detail of the aspect weaver. This clashes with the philosophy of C/C++ programmers. We have illustrated these ideas via a running example implemented in Arachne, tracematches and HALO.

To remedy these issues, we have proposed to apply some key ideas from the HALO pointcut language. HALO contains a limited set of temporal operators, each with a well-known behaviour and memory footprint. A clear connection between operators and the weaver's underlying Rete-engine enables developers to estimate the run-time costs of their pointcuts. We have translated these concepts to an aspect language for C, *Aspicere*, and have illustrated the resulting cHALO system's extensibility by adding a new temporal operator. Currently, we are planning to apply cHALO on a representative case study in the realms of systems software, to measure the real-time memory footprint.

Acknowledgements

The authors want to thank Michael Haupt for encouraging them in designing and implementing cHALO.

8. REFERENCES

- [1] R. Åberg, J. Lawall, M. Südholt, G. Muller, and A.-F. Le Meur. On the automatic evolution of an OS kernel using temporal logic and AOP. In *Proc. of the International Conference on Automated Software Engineering (ASE)*, pages 196–204, October 2003.
- [2] B. Adams and K. D. Schutter. An aspect for idiom-based exception handling (using local continuation join points, join point properties, annotations and type parameters). In *Proc. of the 5th Software-Engineering Properties of Languages and Aspect Technologies Workshop (SPLAT), AOSD*, Vancouver, Canada, March 2007.
- [3] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to aspectj. In *Proc. of the 20th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications (OOPSLA)*, pages 345–364, New York, NY, USA, 2005. ACM.
- [4] P. Avgustinov, J. Tibble, E. Bodden, L. Hendren, O. Lhotak, O. de Moor, N. Ongkingco, and G. Sittampalam. Efficient trace monitoring. In *Proc. of the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (OOPSLA)*, pages 685–686, New York, NY, USA, 2006. ACM.
- [5] P. Avgustinov, J. Tibble, and O. de Moor. Making trace monitors feasible. *SIGPLAN Not.*, 42(10):589–608, 2007.
- [6] E. Bodden, L. Hendren, and O. Lhoták. A staged static program analysis to improve the performance of runtime monitoring. *Proc. of the 21st European Conference on Object-Oriented Programming (ECOOP)*, pages 525–549, 2007.
- [7] E. Bodden, P. Lam, and L. Hendren. Flow-sensitive static optimizations for runtime monitors. Technical Report abc-2007-3, Sable Research Group, McGill University, July 2007.
- [8] R. Douence, P. Fradet, and M. Südholt. A framework for the detection and resolution of aspect interactions. In *Proc. of the 1st ACM SIGPLAN/SIGSOFT conference on Generative Programming and Component Engineering (GPCE)*, pages 173–188, London, UK, 2002. Springer-Verlag.
- [9] R. Douence, T. Fritz, N. Lorient, J.-M. Menaud, M. Ségura-Devillechaise, and M. Südholt. An expressive aspect language for system applications with arachne. *T. Aspect-Oriented Software Development I*, 3880:174–213, 2006.
- [10] R. Douence and O. Motelet. Sophisticated crosscuts for e-commerce. In *Proc. of the workshop on Advanced Separation of Concerns*, 2001.
- [11] R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. In *Proc. of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (REFLECTION)*, pages 170–186, London, UK, 2001. Springer-Verlag.
- [12] C. L. Forgy. Rete: a fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17–37, September 1982.
- [13] C. Herzeel, K. Gybels, and P. Costanza. Escaping with future variables in halo. In *Proc. of the 7th International Workshop on Runtime Verification (RV)*, volume 4839 of *Lecture Notes in Computer Science*, pages 51–62. Springer, 2007.
- [14] C. Herzeel, K. Gybels, P. Costanza, C. D. Roover, and T. D'Hondt. Forward chaining in halo: An implementation strategy for history-based logic pointcuts, August 2007.
- [15] K. Ostermann, M. Mezini, and C. Bockisch. Expressive pointcuts for increased modularity. In *Proc. of the European Conference on Object-Oriented Programming (ECOOP)*, 2005.