# On the Semantics of Matching Trace Monitoring Patterns

Pavel Avgustinov, Julian Tibble, Oege de Moor

Programming Tools Group
Oxford University, United Kingdom

**Abstract.** Trace monitor specifications consist of a pattern that is matched against the trace of events of a subject system. We investigate the design choices in defining the semantics of matching patterns against traces.
Some systems use an *exact-match* semantics (where every relevant event must be matched by the pattern), while others employ a *skipping* semantics (which allows any event to be skipped during matching). The semantics of exact-match is well established; here we give a semantics to skipping also by providing a translation to exact-match. It turns out the translation is *not* surjective: a pattern language with skipping semantics is strictly less expressive than one with exact-match semantics. That proof suggests the addition of a novel operator to a skipping language that makes it equivalent to exact-match.
Another design decision concerns the atoms in patterns: are these unique runtime events, or can multiple atoms match the same runtime event? Many specification formalisms chose predicates for atoms, and then overlap is natural. There are some exceptions, however, and we examine the consequences of that design choice in some depth.

## 1 INTRODUCTION

In recent years, much research interest has centered on so-called *trace monitors*. The idea is simple, but powerful: a base program's execution is observed by a separate entity (the monitor), and whenever the sequence of events matches some predefined criterion, extra code is executed.

This kind of feature is useful in many situations. For example, when debugging a program, it is possible to use trace monitoring to pinpoint the exact time when something goes wrong, or even take some recovery action. One can use it to implement protocols [13] in a high-level declarative fashion. Many runtime verification concerns also have natural representations as trace specifications, either by specifying all "correct" traces (and taking action when a non-matching trace is encountered), or by specifying violating traces (and reacting to successfully completed matches). Examples of such properties are various API contracts and project-specific rules that ideally would be automatically enforced, but aren't checkable by a conventional compiler.

Investigations of trace monitoring have originated from two distinct communities. Aspect-oriented programming (AOP) is a paradigm that allows additional code to be triggered when certain *joinpoints* (identifiable events during program execution) are encountered. From this perspective, trace monitoring is a generalisation of the matching mechanism from single joinpoints to *sequences* of joinpoints, and a number of implementations have arisen in this area (*e.g.* [1,6,8,13]). Largely

independently, the usefulness of program traces as an easily available representation of program behaviour has been exploited by the Runtime Verification (RV) community to implement several tools that check various constraints using this technique (*e.g.* [3, 5, 10, 11]).

Trace monitoring is a useful and desirable feature, as evidenced by the sheer number of systems implementing it. A wide variety of formalisms for describing the patterns of interest are represented: from regular expressions [1] over context-free languages [13] to temporal logics [5] and custom query languages [10, 11]; even a general logic-independent framework [6] has been proposed. Most systems follow the intuition above by running concurrently with the base program and updating matching state as runtime events are encountered, but some [6] also support an *offline* matching mode, which dumps all events to a log file that can be analysed after the program run is complete.

Still, there are several important issues on which the various systems differ. One of them is the question of whether the trace patterns can be parameterised by *free variables*. This allows the programmer to write patterns that are quantified over cliques of interacting objects, but they significantly complicate the semantics and implementation of a trace monitoring feature [**?**], and so they are often not supported.

Another important aspect in which the above approaches differ is the style of semantics: some [1, 8, 13] use what we call an *exact-match semantics*, while others [5, 10, 11] use a more liberal *skipping semantics*. The distinction is that under an exact-match interpretation, each of the events of interest picked out by the monitor must be accounted for by the pattern: using the intuition suggested by temporal logics, the pattern $AB$ means "$A$ and next $B$". In contrast to this, a skipping semantics allows an arbitrary number of events to be ignored in between matched events: $AB$ means "$A$ and eventually $B$". To date, only *tracematches* have provided a formal and exhaustive account of the matching process including free variables [1], but since they are an exact-match feature, this means that there has been no formal evaluation of the implications of a skipping interpretation for a trace monitor.

In this paper, we aim to address this by examining the language design choices that need to be made when implementing a skipping feature. We start with a summary of our previous work on the semantics of tracematches in Section 2; in particular, we give a general technique for reducing the problem of matching a pattern with free variables to the traditional problem of matching a *ground* pattern. In Section 3, we investigate the relationship of exact-match and skipping languages in detail, by defining a small skipping language, giving a semantics and exploring its expressiveness. Semantic issues regarding definitions of symbols and events are described and addressed in Section 4. We give a brief survey of related work in Section 5 and conclude with an outlook on future work in Section 6.

## 2 TRACE PATTERNS WITH FREE VARIABLES

As the starting point of our investigation, we shall summarise our previous work on the semantics of *tracematches*, as introduced in [1]. One of the main design considerations for that particular system was tight integration with the AspectJ

language. As such, events of interest are defined using the standard AspectJ mechanism for selecting runtime events (*pointcuts*), and an entire tracematch can be thought of as a more expressive pointcut triggering an advice body as usual.

As a running example, we shall take a trace monitoring-based implementation of the well-known Observer pattern. There, a particular type of Observer object can register with a certain Subject (captured by the pcCreateObserver pointcut); subsequently any update to the Subject's state (pcUpdateSubject) should trigger a notification of each registered observer. Observers also have the opportunity to deregister with a subject to stop receiving notifications (pcRemoveObserver).

```
1  tracematch(Observer o, Subject s) {
2    sym createObserver after : pcCreateObserver(o, s);
3    sym updateSubject after : pcUpdateSubject(s);
4    sym removeObserver after : pcRemoveObserver(o, s);
5
6    createObserver updateSubject+
7
8    { o.subjectUpdated(s); }
9  }
```

**Fig. 1.** The Observer pattern as a tracematch

The tracematch implementing this is shown in Figure 1. It has three distinct parts: an *alphabet* of symbols (three in this case — lines 2–4), a *regular expression* over these symbols (line 6), and a *body* that is triggered on every successful match (line 8). Each symbol is defined by a standard AspectJ pointcut; moreover, the tracematch header (line 1) can declare free variables that are bound by the symbols and can be used in the body.

Intuitively, the pattern here says "We see an observer register for a particular subject, followed by one or more updates to that subject (and the observer hasn't been disassociated in the meantime)". Each time this matches, we execute the body, which makes use of the acquired bindings to notify the observer.

To perform the matching, we proceed as follows: each runtime event (or *joinpoint*) is matched against every symbol defined by the tracematch; the result is either a failure (in which case the joinpoint is ignored), or a successful match, which may result in some tracematch variables being bound by that symbol. We refer to the transition from the sequence of joinpoints to the sequence of matched symbols annotated with variable bindings as *filtering* the trace, as it essentially discards all events that the tracematch is not interested in.

The careful reader will have noticed that the symbols occurring in the actual tracematch pattern are *ground* — *i.e.* they do not have variable annotations. Matching a sequence of *ground* symbols against the pattern is a well-understood problem: it is just traditional regular expression matching; still, we need to obtain ground traces from the filtered trace.

Thus, after filtering the semantics performs a step called *instantiation*: For each possible valuation (that is, set of runtime values for the tracematch formal variables), the filtered program trace is projected onto the symbols compatible with that valuation. Of course, there are infinitely many possible valuations, but most of

them would result in an empty projection (the projection will only be non-empty if events with matching bindings have occurred in the program history, and there are only finitely many of those at any time). However, each non-empty projection gives us a $(valuation, groundsymboltrace)$ pair, which we can match against the pattern in the usual way. The tracematch body is executed once for each distinct valuation that leads to a successful match.

To illustrate, consider the Observer tracematch with this sequence of events: sub.addObserver(o1); sub.foo(); sub.addObserver(o2); sub.bar(); sub.delObserver(o2); sub.update(); (here sub is a subject, and o1 and o2 are observer objects).

Filtering would discard the irrelevant events s.foo() and s.bar(), and match each of the remaining events with each symbol, resulting in the following *filtered trace*: createObserver(o1, sub); createObserver(o2, sub); removeObserver(o2, sub); updateSubject(sub).

Now, there are two valuations that result in non-empty projections: $(o = o1 \wedge s = sub)$ and $(o = o2 \wedge s = sub)$. The respective instantiated traces are just createObserver updateSubject for the former and createObserver removeObserver updateSubject for the latter. Only the former matches the pattern, so we execute the tracematch body with the first valuation, notifying o1 but not o2.

At this stage, it should be clear why we defined the additional symbol removeObserver in Figure 1, even though it wasn't used in the pattern. Had we omitted it, any calls to the removeObserver() method would have been filtered out without affecting the matching state, and so observers would have still received update notifications after they had been deregistered. The extra symbol ensures those method calls remain in the filtered, instantiated traces; since it is not part of the pattern, no sequence of events that contains it will trigger the tracematch body.

Note that the two-step process of filtering followed by instantiation gives a general strategy for reducing the semantics of matching a trace pattern with free variables to the simpler problem of matching a ground pattern against a ground trace.

We have now summarised our denotational semantics for tracematches [1]. The (equivalent) operational semantics suggests two restrictions that need to be placed on patterns so that they are useful in practice: no pattern should accept the empty sequence, and any trace that matches the pattern must guarantee all free variables bound. In the case of regular expressions, this amounts to checking that every word in the language of the regex contains at least one binding symbol for every variable. Also, some care needs to be taken when updating monitors that aren't fully bound; a full exploration of this issue is beyond the scope of the present document, and we refer the interested reader to [1].

## 3 INTERPRETATION OF PATTERNS

Trace monitoring systems have been developed by researchers in the fields of Aspect-Oriented Programming [1, 8, 13] and Runtime Verification [3, 5, 10, 11]. It is interesting to observe that there seems to be a division regarding the syle of semantics used: systems with an AOP background tend to employ *exact-match semantics*, while those with roots in RV predominantly use *skipping*. In this section, we will examine the differences in-depth, define a small skipping language and proceed to give the first formal semantics of such a language.

The traditional interpretation of many language formalisms (particularly regular and context-free patterns) is essentially exact-match: every symbol in an input word must be accounted for by the pattern; if there is an incompatible symbol then there is no match. In the case of tracematches, this is ensured by the filtering and instantiation steps. Perhaps due to a stronger influence from temporal logics, skipping patterns are not concerned with the precise sequence of events; rather, they specify events that happen one after the other, with any events allowed in between. Arguably, this leads to nicer specifications for many RV properties, as essentially a pattern can be a *least counter-example* to the concern being checked.

### 3.1 A Simple Skipping Language

| | | | |
|---|---|---|---|
| pattern | := term | \| | seqPattern '\|' pattern |
| term | := simplePattern | \| | simplePattern term |
| | | \| | simplePattern '∗' |
| simplePattern | := symbol | \| | '(' pattern ')' |

**Fig. 2.** Regular expression patterns in tracematches

In order to compare the two kinds of pattern interpretation systematically, we will define two related languages. The first, which we shall call $\mathcal{L}_{\text{em}}$, is just the fragment of the tracematch language obtained by omitting the + operator, as it adds no expressive power over the standard Kleene star $*$. This language will be interpreted in an exact-match style, and as tracematches have a formal semantics, $\mathcal{L}_{\text{em}}$ inherits it without modification. Figure 2 illustrates its syntax.

The second language, $\mathcal{L}_{\text{skip}}$, is a skipping language that is as close as possible to the tracematch formalism — that is, it has an explicit alphabet of symbols that may bind free variables, and the pattern is regex-like.

A number of design considerations need to be addressed, however. Firstly, note that any skipping language must have explicit negation; in its absence it would be impossible to prohibit certain events from occurring. We borrow PQL's syntax [11] and write ∼A for the negation of a symbol A.

It is important to understand this construct fully in order to see why the restrictions we shall place upon it are necessary. This is *not* the usual complementation, as found in Extended Regular Expresions (EREs): When we write a pattern like A ∼B A, we do not mean "we see an A, followed by any sequence not matching the pattern B, followed by another A" (as would be the ERE interpretation; this would match the trace *a b b a*). Instead, the negation constrains the set of symbols we are allowed to skip. Thus, the above pattern really means "We see an A, then any number of symbols that aren't B, followed by a second A" (and does *not* match the trace *a b b a*).

The distinction is important: it is well-known that EREs can be transformed into standard regular expressions (at the cost of a non-elementary blowup in the size of the pattern), so the semantics of an exact-match ERE language (as implemented, for example, by one of JavaMOP's logic plugins [6]) do not differ significantly from the standard tracematch semantics. In a skipping setting, negation isn't matched against a part of the trace directly, but affects how the subpatterns around it are matched.

Because of this, it is not well-defined to use negation on compound patterns: The pattern ∼(A B) (while perfectly valid as an ERE pattern) makes no sense in a skipping setting. Indeed, it is common for skipping features to restrict negation — in the case of PQL [11], it is only allowed on single symbols. We will go beyond that, and allow negation of *alternations of symbols*, or, equivalently, of *symbol sets*. Thus, in our small language, it is legal to write ∼(A|B), and this allows skipping of only those events that do not match A or B.

An additional pitfall is sequential composition of patterns. Consider the following example: ∼A ∼B. This is ill-defined in the same way that ∼(A B) is ill-defined; it makes no sense with our interpretation of negation. As a consequence, we must also restrict sequential composition to prevent this kind of problem from occurring. We achieve this by allowing composition only on *fenced terms* (which are just terms that do not begin or end in negation).

| | | |
|---|---|---|
| pattern := fencedTerm | | fencedTerm **'\|'** pattern |
| fencedTerm := simplePattern | \| | simplePattern fencedTerm |
| | \| | simplePattern **'∼'** symbolSet fencedTerm |
| simplePattern := symbol | \| | **'('** pattern **')'** |
| symbolSet := symbol | \| | **'('** symbol **'\|'** symbolSet **')'** |

**Fig. 3.** Syntax of patterns in $\mathcal{L}_{\text{skip}}$

Finally, the overall pattern should also be a fenced term. The resulting skipping language, $\mathcal{L}_{\text{skip}}$, is shown in Figure 3.

Note that Kleene closure is *not* present in $\mathcal{L}_{\text{skip}}$. This is not an oversight on our part; indeed, it turns out that the traditional interpretation of the Kleene star is redundant under a skipping semantics. Suppose $t$ is a fenced term. We can expand the Kleene closure $t*$ thrice (using alternations), and then observe that *any* sequence of events is allowed in between the two symbols $tt$ in the penultimate disjunct, in particular $t*$.

$$\begin{aligned} t* &\equiv \varepsilon \mid t \mid t\ t \mid t\ (t*)\ t \\ &\equiv \varepsilon \mid t \mid t\ t \end{aligned}$$

### 3.2 Semantics of $\mathcal{L}_{\text{skip}}$

We are now ready to give a formal semantics for $\mathcal{L}_{\text{skip}}$, the skipping language defined above. This will take the form of a syntax-directed translation of terms in the skipping language $\mathcal{L}_{\text{skip}}$ to equivalent patterns in our tracematch language $\mathcal{L}_{\text{em}}$. As the semantics of matching $\mathcal{L}_{\text{em}}$ patterns are well-understood, this procedure will capture the meaning of $\mathcal{L}_{\text{skip}}$ patterns.

$$\begin{aligned}
&[\![\ \text{fTerm} \mid \text{pat}\ ]\!] &&\Longrightarrow [\![\ \text{fTerm}\ ]\!] \ \mid\ [\![\ \text{pat}\ ]\!] \\
&[\![\ \text{sPat fTerm}\ ]\!] &&\Longrightarrow [\![\ \text{sPat}\ ]\!]\ \Sigma*\ [\![\ \text{fTerm}\ ]\!] \\
&[\![\ \text{sPat} \sim\alpha\ \text{fTerm}\ ]\!] &&\Longrightarrow [\![\ \text{sPat}\ ]\!]\ (\Sigma \setminus \alpha)*\ [\![\ \text{fTerm}\ ]\!] \\
&[\![\ \text{symbol}\ ]\!] &&\Longrightarrow \text{symbol}
\end{aligned}$$

**Fig. 4.** Translation from $\mathcal{L}_{\text{skip}}$ to $\mathcal{L}_{\text{em}}$

The rewrite rules are given in Figure 4. We can see that alternation in $\mathcal{L}_{\text{skip}}$ is translated to alternation in $\mathcal{L}_{\text{em}}$; any particular symbol is also translated to itself. The interesting cases involve sequential composition (with or without negation). If we have a simple pattern followed by some fenced term, we evaluate this by translating the pattern, inserting $\Sigma*$ and translating the fenced term. Recall that $\Sigma$ is the set of all symbols defined by the tracematch; thus the subpattern $\Sigma*$ amounts to allowing an arbitrary sequence of events, whether or not they are matched by some symbol, to occur.

Consider now negation. If a symbol set $\alpha \subseteq \Sigma$ is prohibited between a simple pattern and a fenced term in $\mathcal{L}_{\text{skip}}$, then the translation again recursively rewrites the pattern, then allows an arbitrary sequence of events that match symbols in $\Sigma \setminus \alpha$, followed by a sequence matching the fenced term. Note how the intended meaning of $[\![$ sPat fTerm $]\!]$ (allow any events in between the two subexpressions) and $[\![$ sPat $\sim\alpha$fTerm $]\!]$ (allow any event except those in $\alpha$ between the two subexpressions) become fully explicit after the translation.

In order for the semantics of a pattern to be well-defined, recall that it needs to bind all free variables. It is not difficult to see that (based on the translation above) an $\mathcal{L}_{\text{skip}}$ pattern is permissible if and only if its translation is permissible in $\mathcal{L}_{\text{em}}$.

### 3.3 Expressiveness of $\mathcal{L}_{\text{skip}}$

The fact that we found a meaning-preserving embedding of $\mathcal{L}_{\text{skip}}$ into $\mathcal{L}_{\text{em}}$ shows that our skipping language is no more expressive than the exact-match language. In other words, merely interpreting a formalism with the skipping paradigm does not increase the expressive power. Recall, however, that we were forced to exclude Kleene closure from $\mathcal{L}_{\text{skip}}$. It is natural to presume that in accommodating the skipping paradigm, we were forced to make our language less expressive.

In this section, we will show that this is indeed the case. We will do this by providing a backwards translation — a syntax-directed rewriting of $\mathcal{L}_{\text{em}}$ patterns into $\mathcal{L}_{\text{skip}}$— for a proper subset of $\mathcal{L}_{\text{em}}$. Indeed, the fragment we will consider contains all translations produced by the rewriting presented in Section 3.2, and will therefore be *equivalent* in expressive power to $\mathcal{L}_{\text{skip}}$.

To obtain a suitable fragment of $\mathcal{L}_{\text{em}}$, it turns out we need to restrict Kleene closure: we will allow only patterns in which the Kleene star is applied to a symbol set, rather than to a general pattern. Moreover, no two Kleene-starred expressions must appear next to each other. Note how these restrictions mirror the restrictions on negation and sequential composition that we were forced to make in designing $\mathcal{L}_{\text{skip}}$.

$$
\begin{array}{ll}
[\![ \text{ term} \mid \text{pat } ]\!] & \Longrightarrow [\![ \text{term} ]\!] \mid [\![ \text{pat} ]\!] \\
[\![ \text{ sPat term } ]\!] & \Longrightarrow [\![ \text{sPat} ]\!] \sim\Sigma [\![ \text{term} ]\!] \\
[\![ \text{ sPat } \alpha* \text{ term } ]\!] & \Longrightarrow [\![ \text{sPat} ]\!] \sim(\Sigma \setminus \alpha) [\![ \text{term} ]\!] \\
[\![ \text{ symbol} ]\!] & \Longrightarrow \text{symbol}
\end{array}
$$

**Fig. 5.** Translation from a fragment of $\mathcal{L}_{\text{em}}$ to $\mathcal{L}_{\text{skip}}$

The translation is shown in Figure 5. Again, alternation is mapped to alternation and symbols to themselves. When translating the sequential composition of

a pattern and a term, we insert the negation of $\Sigma$ between the recursively translated subexpressions — this achieves precisely the desired effect of prohibiting all events matching a declared symbol at that point, as specified by an exact-match semantics. Finally, if there is a Kleene closure of a symbol set $\alpha$ in between two subpatterns, then we insert the negation of the symbol set $\Sigma \setminus \alpha$ — that is, we prohibit everything that is not in $\alpha$, and hence allow everything that *is* in $\alpha$.

This completes the backwards translation; we have proved $\mathcal{L}_{\text{skip}}$ equivalent to a *proper subset* of our tracematch language $\mathcal{L}_{\text{em}}$. The fact that skipping makes traditional Kleene closure obsolete prevents $\mathcal{L}_{\text{skip}}$ from expressing patterns like A(BC)$*$ — which matches all words that start with an A, followed by any number of BC-pairs with no other symbols interspersed.

Interestingly, translating the exact-match pattern from the Observer tracematch (Figure 1) does not yield the skipping pattern one would naturally write: createObserver ∼removeObserver updateSubject. Applying the backward translation to the $\mathcal{L}_{\text{em}}$ pattern (rewritten to use Kleene star), yields the $\mathcal{L}_{\text{skip}}$ pattern: createObserver ∼(createObserver|removeObserver) updateSubject.

### 3.4 Making $\mathcal{L}_{\text{skip}}$ more expressive

A natural consideration after the discussion above is the question of how we can "fix" the expressiveness of $\mathcal{L}_{\text{skip}}$— what kind of feature would we need to add to be able to express the same patterns as $\mathcal{L}_{\text{em}}$?

To give a satisfactory answer, recall that the reason we lost some expressive power was in subtle interactions between negation (as interpreted in a skipping setting) and Kleene closure. This suggests trying an extension that combines the two in some way to achieve the desired results.

As one possibility, we propose a new kind of Kleene closure-like operator, which is parameterised by a set of symbols $\alpha$: $\oplus_\alpha$. It can be applied to any fenced term, and the intended interpretation is:

$$t\oplus_\alpha \quad \equiv \quad t \;\mid\; t \sim\alpha \;\; t\oplus_\alpha$$

Intuitively, $t\oplus_\alpha$ is the Kleene closure of $t$ obtained while prohibiting all events in $\alpha$ between the repetitions of $t$. Note that $t\oplus_\alpha$ is a *fenced* term.

First of all, we shall augment the $\mathcal{L}_{\text{skip}}$ semantics with a translation rule for our new operator:

$[\![\ \text{term}\oplus_\alpha]\!] \Longrightarrow ([\![\ \text{term}\ ]\!]\ (\Sigma\backslash\alpha)*)*\ \ [\![\ \text{term}\ ]\!]$

Intuitively, this says: match zero or more occurrences of "a trace matching term followed by any symbols not in $\alpha$", followed by a trace matching term.

Finally, we will complete our examination by translating unrestricted Kleene closure in $\mathcal{L}_{\text{em}}$ into our extended skipping language. First of all, we rewrite the exact-match pattern (using alternations if necessary) in such a way that all occurrences of the Kleene star are replaced by the Kleene "plus" (unless they are on symbol sets, in which case the old translation already accounts for them). Then we can handle the resulting terms as follows:

$[\![\ \text{term}+\ ]\!] \Longrightarrow [\![\text{term}\ ]\!]\oplus_\Sigma$

Recall the example pattern $A(BC)*$ that, as we argued above, can't be captured by the original skipping language. With the extension, it just becomes the following pattern: $A \mid A(B \sim(\Sigma)C)\oplus_\Sigma$.

## 4 DEFINITION OF SYMBOLS AND EVENTS

Another way in which semantics of trace-monitors vary is in the relation between *symbols* and *runtime events*. To clarify, by "runtime event" we mean a joinpoint, *i.e.* a well-defined point during program execution. Such an event may or may not be relevant to a trace monitor; this is determined by matching it against the *symbols*. Now, unlike in traditional formal language theory, in this context symbols may *overlap* — that is, two or more different symbols could match the same runtime event.

One possible interpretation is suggested by temporal logics: we can view symbols as predicates that may or may not hold at the same point in time (joinpoint). If the pattern language supports it, one could then write specifications that require both symbol $A$ and symbol $B$ to hold at the same time, and indeed approaches that are based on temporal logics often make use of Boolean combinations of symbols. Tracematches don't allow this, but fully support overlapping symbols. The interpretation there is that if several symbols match a runtime event, then there are several filtered traces, one for each possible symbol instantiation.

An alternative view is to treat the trace monitor as a consumer of *monitor events*, which are different than runtime events in that they correspond to exactly one symbol. They are different from symbols because they may have associated variable bindings. This means that if a runtime event matches multiple symbols, then all of these instantiated symbols occur (as monitor events) in the filtered trace in some order.

To illustrate the difference, we will compare the JavaMOP and Tracematch systems on a small example, because their symbol definitions look very similar — both have adapted the pointcut language of AspectJ for this purpose. Two symbols are declared for each system in Figure 6. In both cases, 'Clear' is defined as 'immediately before calling a method named *clear*', and 'List' is defined as 'immediately before calling a method on an object of type *List*'.

| JavaMOP: | Tracematches: |
|---|---|
| **Event** Clear : **begin**(**call**(∗ ∗.clear()));<br>**Event** List : **begin**(**call**(∗ List .∗(..) ));| **sym** Clear **before** : **call**(∗ ∗.clear());<br>**sym** List **before** : **call**(∗ List .∗(..) ); |

**Fig. 6.** Symbol declarations for JavaMOP and Tracematches

However, since these symbols overlap, the respective patterns will sometimes be matched against different traces. Consider for example this sequence of calls: someSet.clear(); someList.clear(); someList.size(), in which the first call matches the symbol *Clear*, the second matches both symbols, and the last matches *List*.

The difference is, of course, caused by the second call. Tracematches interpret symbols as predicates; correspondingly, there are two filtered traces to match against: *Clear Clear List* and *Clear List List*. In JavaMOP, on the other hand, the second call gives rise to *two* atomic, distinct events, so that the pattern is matched against *Clear Clear List List*.

Interpreting symbols as events is problematic because the number of symbols in a program trace need not the same as the number of joinpoints in the base program. In the example above, the pattern (Clear|List)(Clear|List)(Clear|List)

might be expected to match at the third call, but it would in fact also match at the second call. It is also difficult to justify the ordering of events when they are triggered by a single joinpoint. For example, why is the trace not *Clear List Clear List*? The JavaMOP designers resolved this non-determinism by specifying that events corresponding to the same cause are generated in the order that the symbols are defined in the specification.

# 5 RELATED WORK

| System | Implementation | | Semantic choices | |
|---|---|---|---|---|
| | free variables | formalism | exact-match | overlapping |
| tracematches [1] | yes | regex | yes | predicates |
| PQL [11] | yes | CFG | no | predicates |
| PTQL [10] | yes | SQL | no | predicates |
| JavaMOP [6] | partial | multiple | yes | events |
| J-LO [12] | yes | LTL | no | predicates |
| tracecuts [13] | no | CFG | yes | events |
| HAWK [7] | yes | multiple | yes | predicates |
| Arachne [9] | yes | regex | n/a | predicates |

**Table 1.** Overview of some trace monitoring systems

The purpose of the present paper is to relate multiple strands of research, in particular contrasting the matching semantics used in aspect-oriented programming and those in mainstream runtime verification. We have furthermore pointed out subtle variations that exist between different systems in the definition of symbols and events. So far our discussion has focussed on concepts; here we briefly relate those concepts to systems that instantiate them.

Table 1 lists some systems and, where applicable, the choices made in their design. These are often conscious choices, but sometimes the result of an accident of implementation. Concretely, we examine whether the style of semantics is exact-match or skipping, and how overlapping symbols are handled (if they are possible to define). For completeness, we also state the formalism each system uses, and whether or not free variables are supported.

PQL [11] is the principal example of a system with skipping semantics. Its patterns may be recursively defined, which is why we marked the formalism as Context Free Grammars (CFG) in the above table.

PTQL [10] also warrants some further discussion. It uses an SQL like language to write patterns, and joinpoints are regarded as records that have fields for (logical) start and end times. So to say that $e_1$ occurred before $e_2$, one states the condition $e_1.endTime < e_2.startTime$ in the query. With such a formalism, skipping is clearly the semantics of choice. Consequently the language does not provide for a Kleene star.

JavaMOP [6] is particularly interesting and relevant, because of its goal to be a framework where different specification formalisms can be plugged in. Indeed, such plugins can support both extended regular expressions and LTL variants. It supports free variables, but only as a matching context: the values bound by the variables aren't available for use when a match is completed (it is, however, possible to use *action blocks* associated with symbols to manually track bindings).

J-LO [12] represents a blend of traditional runtime verification (it uses LTL to express patterns), and aspect-oriented programming (events are specified with AspectJ pointcuts). With the use of LTL, and the focus on finding violations of properties, skipping semantics is the natural choice.

Tracecuts [13] is by contrast squarely in the aspect-oriented programming tradition. It uses an exact match semantics for context-free patterns. It takes, however, a different approach from tracematches in that events cannot overlap.

HAWK [7] is perhaps the most general formalism discussed here: it provides a rule based system that can express temporal logics, metric logics, extended regular expressions and it allows the user to define new logics. We hope that the present paper helps to map out the choices that designers of such logics must face.

Arachne [9] is a trace monitoring system for C. Its specification formalism is like a process algebra, restricted to the linear recursions that correspond to finite automata - hence we classified it as 'regular expressions'. The specifications are however more like definitions in process algebra, and hence the issue of skipping versus exact match does not arise: the matching process is coded by hand.

We have omitted systems from Table 1 that are so different as to be incomparable. For example, in Alpha [4], the queries are written as Prolog predicates over lists of events. Again, at that level of specification, there is no distinction between skipping and non-skipping: the programmer has to control such behaviour herself. Naturally that makes Alpha extremely expressive, but such expressivity comes at the cost of much poorer execution speed than the other systems compared here.

## 6 CONCLUSION AND FUTURE WORK

The contributions of this paper are as follows:

- Reduce the semantics of matching with variables to matching without variables, via the two-sep process of filtering and instantiation.
- A careful analysis of what pattern language constructs make sense for the skipping semantics: especially negation needs special care, as does Kleene closure.
- Pin down the semantics of skipping via a translation to exact-match, which already has a well understood semantics.
- Demonstrate that this translation is not onto, and therefore skipping is strictly less expressive than exact match.
- Propose a new pattern language construct (a modified form of Kleene closure) that, when included in a skipping language, makes it equivalent in expressive power to an exact-match language.

These contributions are an encouraging start to the comparative study of the semantics of matching patterns to traces, but much remains to be done. A pressing question concerns the inclusion of recursion (context-free languages) in the pattern specifications. We have seen that in the treatment of Kleene closure some subtleties arose, and it will be interesting to generalise that study to arbitrary recursive patterns.

Ultimately, it is our aim to provide a unifying framework for all these formalisms based on Datalog. In a separate strand of research, we have described a semantics of the AspectJ pointcut matching mechanism in terms of Datalog [2],

and we are now extending that to richer formalisms for runtime verification, including those discussed here. We plan to use such a semantics in terms of Datalog as the basis of a single unified backend for many runtime verification systems.

## References


1. Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding Trace Matching with Free Variables to AspectJ. In *Object-Oriented Programming, Systems, Languages and Applications*, pages 345–364. ACM Press, 2005.
2. Pavel Avgustinov, Elnar Hajiyev, Neil Ongkingco, Oege de Moor, Damien Sereni, Julian Tibble, and Mathieu Verbaere. Semantics of static pointcuts in aspectj. In Matthias Felleisen, editor, *Principles of Programming Languages (POPL)*. ACM Press, 2007.
3. Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Rule-based runtime verification. In *Fifth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 04)*, volume 2937, pages 44–57. Lecture Notes in Computer Science, 2003.
4. Christoph Bockisch, Mira Mezini, and Klaus Ostermann. Quantifying over dynamic properties of program execution. In *2nd Dynamic Aspects Workshop (DAW05)*, Technical Report 05.01, pages 71–75. Research Institute for Advanced Computer Science, 2005.
5. Eric Bodden. J-LO - A tool for runtime-checking temporal assertions. Master's thesis, RWTH Aachen University, 2005.
6. Feng Chen and Grigore Roşu. Java-MOP: A monitoring oriented programming environment for Java. In *11th International Conference on Tools and Algorithms for the construction and analysis of systems (TACAS '05)*, volume 3440 of *Lecture Notes in Computer Science*, pages 546–550. Springer Verlag, 2005.
7. Marcelo d'Amorim and Klaus Havelund. Event-based runtime verification of java programs. In *WODA '05: Proceedings of the third international workshop on Dynamic analysis*, pages 1–7. ACM Press, 2005.
8. Rémi Douence, Thomas Fritz, Nicolas Loriant, Jean-Marc Menaud, Marc Ségura, and Mario Südholt. An expressive aspect language for system applications with arachne. In *Aspect-Oriented Software Development*, pages 27–38. ACM Press, 2005.
9. Thomas Fritz, Marc Ségura, Mario Südholt, Egon Wuchner, and Jean-Marc Menaud. An application of dynamic AOP to medical image generation. In *2nd Dynamic Aspects Workshop (DAW05)*, Technical Report 05.01, pages 5–12. Research Institute for Advanced Computer Science, 2005.
10. Simon Goldsmith, Robert O'Callahan, and Alex Aiken. Relational queries over program traces. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 385–402. ACM Press, 2005.
11. Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors using PQL: a program query language. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 365–383. ACM Press, 2005.
12. Volker Stolz and Eric Bodden. Temporal Assertions using AspectJ. In *Electronic Notes in Theoretical Computer Science*, volume 144, pages 109–124, 2006.
13. Robert Walker and Kevin Viggers. Implementing protocols via declarative event patterns. In *ACM Sigsoft International Symposium on Foundations of Software Engineering (FSE-12)*, pages 159–169. ACM Press, 2004.