On the Semantics of Matching Trace Monitoring Patterns

Pavel Avgustinov Oege de Moor Julian Tibble
[pavel.avgustinov, oege.de.moor, julian.tibble]@comlab.ox.ac.uk
Programming Tools Group, University of Oxford, United Kingdom

1. INTRODUCTION

Many runtime verification properties can be expressed as a pattern that is matched against the trace of runtime events. In our previous work on so-called tracematches, the pattern language consists of regular expressions over events; furthermore the regular expressions contain free variables [1]. The latter feature is indispensable when tracking the behaviour of a clique of objects. The formal semantics of tracematches was worked out in [1]: that paper both gives a declarative semantics and proves it is equivalent to an operational semantics, which in turn is the basis of an efficient implementation.

When comparing tracematches against other formalisms in runtime verification (for instance the PQL system [3]), an important difference is the use of an *exact-match* semantics (where every relevant event must be matched by the pattern) instead of a *skipping* semantics (any event may be skipped during matching). Under skipping semantics, '$A\,B$' means 'A implies eventually B', while under exact-match, '$A\,B$' means 'A implies next B'. It appears that the exact-match semantics are favoured by systems originating in aspect-oriented programming [2, 4, 5], while skipping semantics are more common for systems with roots in runtime verification. Indeed, one could argue that skipping is more natural for specifying temporal properties.

This paper investigates the formal relation between these two styles of specification. In particular, we show that a skipping semantics is no more expressive than exact-match.

2. SEMANTICS OF TRACE PATTERNS

2.1 Exact-match languages

Since the pattern language of *tracematches* [1] is the only one that comes equipped with a formal semantics, we shall make it the starting point of our investigation.

In a nutshell, each tracematch declares a set of symbols, which pick out the events of interest (in fact, they are AspectJ pointcuts). We can think of these as predicates on events, following the intuition suggested by temporal logics. Matching such a predicate (*i.e.* symbol) to an event either results in failure, or in success (possibly subject to a certain instantiation of the tracematch's formal variables).

The pattern itself is a regular expression over the alphabet of symbols, and is matched against *all suffixes* of the *filtered, instantiated trace* — that is, first, the sequence of program events is filtered to a sequence of symbols (*i.e.* propositions that are true at each event), and this is then instantiated for each possible set of variable bindings, resulting in a sequence of *ground symbols*. Since tracematches use an exact-match semantics, matching the regular expression against strings of ground symbols is well-understood.

2.2 Skipping languages

As a first step, we shall provide a formal definition of a simple tracematch-like skipping language: We require an alphabet of declared symbols over which patterns will range, and we shall try to follow the syntax of regular expressions as closely as possible.

It is clear that explicit negation is needed in a skipping language, as otherwise it would be impossible to specify patterns that preclude certain events from occurring in the middle of a match. However, if negation is unrestricted and can apply to general patterns, then a matcher (and hence a semantics) for the language must be capable of arbitrary backtracking, which significantly complicates matters. PQL [3] solves this problem by only permitting negation on a single declared symbol; we go somewhat beyond that by allowing negations of *alternations* of symbols (or, equivalently, symbol sets).

Note that in order to ensure this, we have to use negated terms with care. In particular, they cannot be used freely in sequential composition: since ~a ~b would match exactly the same traces as ~(a b), it is equivalent to a negated compound pattern. We refer to terms that do not start or end in negation as *closed* (cf. Figure 1), and only such terms may be freely sequentially composed.

```
pattern := closedTerm
         | closedTerm '|' pattern

closedTerm := simplePattern
            | simplePattern closedTerm
            | simplePattern '~' symbolSet
                                closedTerm

simplePattern := symbol
               | '(' pattern ')'

symbolSet := symbol
           | '(' symbol '|' symbolSet ')'
```

Figure 1: The grammar of our simple skipping language

It is interesting to observe that the Kleene star is not present in our skipping language definition. This is not an oversight: Under a skipping semantics, a b* c would match

$$[\![\ \text{cTerm} \mid \text{pat}\ ]\!] \Longrightarrow [\![\ \text{cTerm}\ ]\!] \mid [\![\ \text{pat}\ ]\!]$$

$$[\![\ \text{sPat cTerm}\ ]\!] \Longrightarrow [\![\ \text{sPat}\ ]\!]\ \Sigma* [\![\ \text{cTerm}\ ]\!]$$

$$[\![\ \text{sPat}\ \tilde{}\alpha\ \text{cTerm}\ ]\!] \Longrightarrow [\![\ \text{sPat}\ ]\!]\ (\Sigma \setminus \alpha)* [\![\ \text{cTerm}\ ]\!]$$

$$[\![\ \text{symbol}\ ]\!] \Longrightarrow \text{symbol}$$

Figure 2: Rewrite rules translating into the tracematch language

precisely when a c matches, since any events are permitted between a and c. Thus, Kleene closure does not add expressiveness, since the matcher could always choose to match it against the empty program trace, and then skip over an arbitrary sequence.

To specify the semantics of this language, we proceed by providing a set of simple syntax-directed rewriting rules that translate a skipping pattern into a standard tracematch pattern (the semantics of which is well-understood). The translation proceeds by structural induction on the skipping pattern; full details are given in Figure 2 (where cTerm is of type closedTerm, pat is a pattern, sPat is a simplePattern, α is an alternation of symbols, interpreted as a set, and Σ is the set of all declared symbols).

As expected, the basic structure of the pattern carries over. Alternation is mapped to alternation, and each individual symbol is mapped to the same symbol. The interesting cases concern sequential composition: either with or without an intervening negated set of symbols.

In the negation-free case, we want to capture the fact that "an arbitrary number of events of any kind are allowed in between consecutive matched statements" [3]. The natural way to ensure this is to add $\Sigma*$ between the two patterns: Since Σ is the entire alphabet of symbols, this has the desired effect.

When there is a negated set of symbols, the translation is similar, but we allow any number of events matching symbols in $\Sigma\backslash\alpha$ to occur. If we interpret the Kleene closure of the empty set to only match the empty trace, then this has exactly the effect of prohibiting the symbols in α under a skipping semantics.

This simple set of rewrite rules suffices to pin down the semantics of our small skipping language; we therefore conclude that changing the interpretation of patterns from exact-match to skipping does not by itself increase expressiveness.

3. EXPRESSIVENESS OF SKIPPING LANGUAGES

Our translation showed that a regular expressions-based skipping language is not more expressive than the corresponding exact-match language. At the same time, we were forced to do without Kleene closure, since any Kleene-starred expression can be dropped from a skipping pattern without altering matching behaviour.

Note that the claim above is only true if we insist that Kleene-starred terms are not *closed terms* in the sense of the condition on sequential composition imposed above. What would happen if we were to consider them closed?

Unfortunately, the answer is that we end up with a non-compositional semantics, meaning that it is not valid to substitute equivalent subexpressions for each other. Consider the following two patterns: a b* c and a c. They clearly match the same set of traces, so (a b*) and (a) are equivalent subexpressions. Now consider the context C(X) := (X ~b c), defined for all closed instantiations of X. Since both our subexpressions would be closed, we can plug each of them into the context to obtain the patterns a b* b~ c and a ~b c, respectively. But the former matches the trace A B C, while the latter doesn't.

Thus, it would seem that the skipping language is strictly less expressive than the tracematch language. This notion is formalised in the paper by giving a backwards translation from a *subset* of the tracematch language to the skipping language (it turns out that to get an equivalent language, we need to restrict Kleene closure to symbol sets, mirroring the restriction on explicit negation).

4. CONCLUSIONS AND FUTURE WORK

This work examines the semantics of skipping-based trace monitoring languages, and gives a formal semantics for a simple regular expressions-based skipping language, as well as an argument of equivalence to a subset of the trace language of tracematches [1].

Since it was shown that a skipping interpretation precludes the use of Kleene closure, it is interesting to investigate in how far the results presented here carry over when we consider more expressive languages (context-free grammars, or their closure under intersection — this is the class of languages accepted by [3]).

The small trace language described above has been implemented as an extension to the tracematches system and will be made available in the very near future.

5. 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] 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.

[3] 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.

[4] Klaus Ostermann, Mira Mezini, and Christoph Bockisch. Expressive pointcuts for increased modularity. In *ECOOP*, 2005.

[5] 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.