On Matching Concurrent Traces*

Iliano Cervesato[1], Frank Pfenning[2], Jorge Luis Sacchini[1],
Carsten Schürmann[3], and Robert J. Simmons[2]

[1] Carnegie Mellon University – Doha, Qatar
iliano@cmu.edu, sacchini@qatar.cmu.edu
[2] Carnegie Mellon University – Pittsburgh, PA, USA
fp@cs.cmu.edu, rjsimmon@cs.cmu.edu
[3] IT University of Copenhagen – Copenhagen, Denmark
carsten@itu.dk

Abstract. Concurrent traces are sequences of computational steps where independent steps can be permuted and executed in any order. We study the problem of matching on concurrent traces. We outline a sound and complete algorithm for matching traces with one variable standing for an unknown subtrace.

1 State-Transition Concurrency

Computation in a concurrent system results from the interactions of computing units such as threads or agents. One popular approach to modeling concurrent computation views each such agent as being able to perform local transformations on a global state, possibly in parallel. This is the approach embraced by Petri nets [5]. This is also the approach underlying propositional multiset rewriting, which we will now describe in further detail.

A *multiset rewriting system* is determined by a set of *rules* of the form $\tilde{a} \to \tilde{b}$, where $\tilde{a}$ and $\tilde{b}$ are multisets over some support set S. We write "$\cdot$" for the empty multiset and "$\tilde{a}, \tilde{b}$" for the union of multisets $\tilde{a}$ and $\tilde{b}$. Multiset union is associative and commutative, and has the empty multiset as its unit. Therefore, the set of multisets over S is a commutative monoid with respect to "," and "$\cdot$". We give each rule $\tilde{a} \to \tilde{b}$ with a unique name, r, writing the association as $r : \tilde{a} \to \tilde{b}$. We write R for a set of labeled rules.

A *state* s is a set of pairs $x : a$ where $a \in S$ is an element of the support set S and x is a unique name. Abusing notation, we occasionally write such a state as $\tilde{x} : \tilde{a}$, where $\tilde{a}$ is the multiset of occurrences of elements of S in s and $\tilde{x}$ the corresponding names. We also use "," for set union in the context of states.

A rule $r : \tilde{a} \to \tilde{b}$ in a multiset rewriting system R is *enabled* in a state s if $s = s', (\tilde{x} : \tilde{a})$, i.e., if s contains its antecedent. In this circumstance, the

* Partially supported by the Qatar National Research Fund under grant NPRP 09-1107-1-168, the Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) through the Carnegie Mellon Portugal Program under Grant NGN-44, and the Danish Council for Strategic Research, Programme Commission on Strategic Growth Technologies under grant 10-092309.

application of r to s results in the state $s'' = s', (\tilde{y} : \tilde{b})$ where $\tilde{y}$ are fresh names: the portion of s matching $\tilde{a}$ has been replaced with new state elements for $\tilde{b}$. The triple $r(\tilde{x}; \tilde{y})$, generically denoted t, is a rule instance, or *transition.* We call $\tilde{x} : \tilde{a}$ the pre-condition of t and denote it as $\bullet t$ and $\tilde{y} : \tilde{b}$ its *post-condition*, denoted $t\bullet$. We formalize the state transformation embodied by the application of a rule by means of the multiset rewriting judgment $R \vdash s \xrightarrow{t} s''$. Then, application is defined simply as

$$R, (r : \tilde{a} \to \tilde{b}) \vdash \underbrace{s', \tilde{x} : \tilde{a}}_{s} \overset{r(\tilde{x};\tilde{y})}{\longrightarrow} \underbrace{s', \tilde{y} : \tilde{b}}_{s''}$$

or more succinctly as $R \vdash s', \bullet t \xrightarrow{t} s', t\bullet$ for t an instance of a rule $r \in R$.

Multistep computation is obtained by iterating application. We write $R \vdash s \overset{\boldsymbol{t}}{\Longrightarrow} s'$ for the reflexive and transitive closure of our base judgment, where $\boldsymbol{t}$ records the rules that have been applied to go from s to s'. It is defined by the following grammar:

$$\boldsymbol{t} ::= \cdot \ \mid \ r(\tilde{x}; \tilde{y}) \ \mid \ \boldsymbol{t_1}; \boldsymbol{t_2}$$

We call $\boldsymbol{t}$ a *trace*. Here, "$\cdot$" represents the empty trace and "$\boldsymbol{t_1}; \boldsymbol{t_2}$" is the concatenation of $\boldsymbol{t_1}$ and $\boldsymbol{t_2}$.

A concurrent trace $\boldsymbol{t}$ can be interpreted as a bipartite directed acyclic graph (BDAG) $G = (N_1, N_2, E)$ where N_1 is the set of transitions t in $\boldsymbol{t}$ and N_2 is the set of state elements $x : a$ mentioned in $\boldsymbol{t}$. There is an edge from $x : a$ to $t = r(\tilde{x}; \tilde{y})$ if and only if x occurs in $\tilde{x}$, and there is an edge from t to $y : b$ iff y occurs in $\tilde{y}$. These BDAGs are exactly what we get by graphically unfolding the computation of a place/transition Petri net.

2 Concurrent Traces

A trace is a precise record of all the steps that occurred in a concurrent computation. Indeed, given the initial state, it allows us to replay the computation exactly and determine the final state. If the latter is known, it can be replayed backward and reconstruct the initial state.

Traces have an interesting algebraic structure that is key to reasoning about concurrent computations. To expose it, it will be useful to define the natural extension of pre- and post-conditions:

$$\left\{\begin{array}{lcl} \bullet(\cdot) & = & \cdot \\ \bullet r(\tilde{x}; \tilde{y}) & = & \tilde{x} : \tilde{a} \\ \bullet(\boldsymbol{t_1}; \boldsymbol{t_2}) & = & \bullet\boldsymbol{t_1} \cup (\bullet\boldsymbol{t_2} \setminus \boldsymbol{t_1}\bullet) \end{array}\right. \qquad \left\{\begin{array}{lcl} (\cdot)\bullet & = & \cdot \\ r(\tilde{x}; \tilde{y})\bullet & = & \tilde{y} : \tilde{b} \\ (\boldsymbol{t_1}; \boldsymbol{t_2})\bullet & = & (\boldsymbol{t_1}\bullet \setminus \bullet\boldsymbol{t_2}) \cup \boldsymbol{t_2}\bullet \end{array}\right.$$

for $r : \tilde{a} \to \tilde{b}$. Two traces $\boldsymbol{t_1}$ and $\boldsymbol{t_2}$ are *independent*, written $\boldsymbol{t_1} \parallel \boldsymbol{t_2}$, if $\bullet\boldsymbol{t_1} \cap \boldsymbol{t_2}\bullet = \boldsymbol{t_1}\bullet \cap \bullet\boldsymbol{t_2} = \emptyset$. A name x is *internal* to a trace $\boldsymbol{t}$ if it does not occur in neither $\bullet\boldsymbol{t}$ nor $\boldsymbol{t}\bullet$.

Two traces $\boldsymbol{t}$ and $\boldsymbol{t'}$ are *equal*, written $\boldsymbol{t} = \boldsymbol{t}'$, if there are renamings ρ and ρ' of their internal variables such that $R \vdash s \overset{\rho \boldsymbol{t}}{\Longrightarrow} s'$ iff $R \vdash s \overset{\rho' \boldsymbol{t}'}{\Longrightarrow} s'$ for any s and s'. It is easy to prove that if $\boldsymbol{t} = \boldsymbol{t}'$, then $\bullet\boldsymbol{t} = \bullet\boldsymbol{t}'$ and $\boldsymbol{t}\bullet = \boldsymbol{t}'\bullet$.

Trace concatenation is associative with respect to trace equality, and the empty trace ("$\cdot$") is its left and right unit. Moreover, independent traces can be permuted, i.e., $\boldsymbol{t_1};\boldsymbol{t_2} = \boldsymbol{t_2};\boldsymbol{t_1}$ whenever $\boldsymbol{t_1} \parallel \boldsymbol{t_2}$.

Altogether, traces obey the following equational theory:

$$\begin{array}{llcll}
\textit{Associativity} & (\boldsymbol{t_1};\boldsymbol{t_2});\boldsymbol{t_3} & = & \boldsymbol{t_1};(\boldsymbol{t_2};\boldsymbol{t_3}) & \\
\textit{Left identity} & \cdot;\boldsymbol{t} & = & \boldsymbol{t} & \\
\textit{Right identity} & \boldsymbol{t};\cdot & = & \boldsymbol{t} & \\
\textit{Independent commutativity} & \boldsymbol{t_1};\boldsymbol{t_2} & = & \boldsymbol{t_2};\boldsymbol{t_1} & \text{if } \boldsymbol{t_1} \parallel \boldsymbol{t_2}
\end{array}$$

Two traces are equal whenever the corresponding BDAGs are isomorphic up to the name of internal state elements. Although the complexity of trace equality has not been determined as far as we know, the isomorphism problem for both bipartite graphs and directed acyclic graphs is known to be GI-complete, where GI is a complexity class between P and NP [6].

The equational theory we just gave is closely related to trace monoids, the laws that govern Mazurkiewicz trace theory [4]. Mazurkiewicz traces may contain several occurrences of what corresponds to our transitions, while our traces are limited to a single occurrence of each transition.

It should be observed that, although our definition of trace was based on finite concurrent computations, the same equational theory applies to traces of infinite computations, as produced by operating systems, streaming computations, and reactive systems.

3 Reasoning about Concurrent Traces

Reasoning about concurrent systems requires reasoning about concurrent computations, whether implicitly or explicitly. Typical reasoning tasks include 1) the analysis of a given computation, whether to gather information about it as done in profilers or to react to unexpected states as done by monitors; 2) reasoning about all possible computations from a given state, for example to verify correctness as in model checking, to ensure termination, and to detect deadlocks or starvation; and 3) verifying the soundness and often completeness of a program transformation, through some sort of (bi)simulation.

While many of these reasoning tasks can be performed by just looking at the successive states of the system, an analysis that explicitly works on the traces of the concurrent computation can be much more precise and economical, especially when the state is large or geographically distributed. To carry out such forms of trace-based reasoning, we must be able to work with traces that are not completely specified. This can be captured by extending our definition of trace with *trace variables*:

$$\boldsymbol{t} ::= X(\tilde{x};\tilde{y}) \;\mid\; \cdot \;\mid\; r(\tilde{x};\tilde{y}) \;\mid\; \boldsymbol{t_1};\boldsymbol{t_2}$$

Equality and other notions introduced for traces carry over to traces containing variables. They also allow asking whether there are instances of said variables that make two traces equal, a typical unification problem. With the requirement that $\bullet\boldsymbol{t} = \bullet\boldsymbol{t'}$ and $\boldsymbol{t}\bullet = \boldsymbol{t'}\bullet$, we write

$$\boldsymbol{t} \overset{?}{=} \boldsymbol{t'}$$

for such a problem. A solution is a substitution $\theta = (X_1 \leftarrow \boldsymbol{t_1}, \ldots, X_n \leftarrow \boldsymbol{t_n})$ such that for each i, $\bullet X_i = \bullet t_i$ and $X_i\bullet = t_i\bullet$ and moreover $[\theta]\boldsymbol{t} = [\theta]\boldsymbol{t'}$ where substitution application is defined in the standard way.

Because the equational theory of traces is only partially commutative, standard results about ACU or string unification do not apply directly [1]. We have no proof that it is decidable. We know however that there can be multiple solutions to a trace unification problem since multiset unification can be encoded as a form of trace unification where all transitions are independent from each other.

4 Matching

While trace unification is necessary for reasoning tasks (2) and (3) above, matching is sufficient for task (1). In a matching problem $\boldsymbol{t} \overset{?}{=} \boldsymbol{t'}$, the trace $\boldsymbol{t'}$ does not contain trace variables. Matching is decidable: freeze the order of the transitions in $\boldsymbol{t}$ and consider all dependency-preserving permutations of $\boldsymbol{t'}$ in turn; each permutation gives rise to a string matching problem, which is solvable in linear time. Of course such a brute-force approach has exponential complexity.

In this section, we present an algorithm for the restriction of the matching problem with a single trace variable. This algorithm behaves better than the brute force approach just mentioned. These matching problems have therefore the form

$$\boldsymbol{t_1}; X(\tilde{x}; \tilde{y}); \boldsymbol{t'_1} \ \overset{?}{=} \ \boldsymbol{t_2}$$

where $\boldsymbol{t_1}$, $\boldsymbol{t'_1}$ and $\boldsymbol{t_2}$ do not contain trace variables.

Our algorithm will work as follows: strip common transitions from the beginning and the end of the two sides until only the trace variable is left on the left-hand side. While intuitively simple, care must be taken because independent transitions can be permuted and especially because the two sides may not use the same internal names. This means that as we strip initial and/or final transitions, we will need to apply a renaming to the rest of the trace.

Given sequences of names $\tilde{x}$ and $\tilde{x}'$ of the same length, we write $\rho = [\tilde{x}'/\tilde{x}]$ for the renaming that replaces each name x_i in $\tilde{x}$ with the corresponding name x'_i in $\tilde{x}'$. A renaming $\rho = [\tilde{x}'/\tilde{x}]$ is *legal* for a trace $\boldsymbol{t}$ if it is the identity for $\bullet\boldsymbol{t}$ and $\boldsymbol{t}\bullet$. Applying a renaming ρ to a trace $\boldsymbol{t}$, written $\rho\boldsymbol{t}$, preserves $\bullet\boldsymbol{t}$ and $\boldsymbol{t}\bullet$ if ρ is legal for $\boldsymbol{t}$.

Our matching algorithm is given next for a generic problem $\boldsymbol{t_1} \overset{?}{=} \boldsymbol{t_2}$. We fix the order of $\boldsymbol{t_1}$ but allow permutations in $\boldsymbol{t_2}$. We propagate the internal names in $\boldsymbol{t_2}$ to $\boldsymbol{t_1}$. Legality constraints ensure the invariant that $\bullet\boldsymbol{t_1} = \bullet\boldsymbol{t_2}$ and $\boldsymbol{t_1}\bullet = \boldsymbol{t_2}\bullet$.

1. $p(\tilde{x};\tilde{y});\boldsymbol{t_1} \overset{?}{=} p(\tilde{x};\tilde{y}');\boldsymbol{t_2}$:
   If $\tilde{y}'/\tilde{y}$ is legal for $p(\tilde{x};\tilde{y});\boldsymbol{t_1}$, then solve $[\tilde{y}'/\tilde{y}]\boldsymbol{t_1} \overset{?}{=} \boldsymbol{t_2}$, otherwise fail.
2. $\boldsymbol{t_1};p(\tilde{x};\tilde{y}) \overset{?}{=} \boldsymbol{t_2};p(\tilde{x}';\tilde{y})$:
   If $\tilde{x}'/\tilde{x}$ is legal for $\boldsymbol{t_1};p(\tilde{x};\tilde{y})$, then solve $[\tilde{x}'/\tilde{x}]\boldsymbol{t_1} \overset{?}{=} \boldsymbol{t_2}$, otherwise fail.
3. $X(\tilde{x};\tilde{y}) \overset{?}{=} \boldsymbol{t_2}$: Simply return the solution $X \leftarrow \boldsymbol{t_2}$.

This algorithm is sound and complete in the sense that computed solutions do yield equal traces, and that it computes all solutions. Proofs, although using a different presentation of traces, will appear in a forthcoming technical report.

Theorem 1 (Soundness). *Given a matching problem* $\boldsymbol{t_1};X(\tilde{x};\tilde{y});\boldsymbol{t_1'} \overset{?}{=} \boldsymbol{t_2}$*, if the matching algorithm reports a solution* $X \leftarrow \boldsymbol{t}$*, then there is a legal renaming* ρ *such that* $\rho\boldsymbol{t_1};\boldsymbol{t};\rho\boldsymbol{t_1'} = \boldsymbol{t_2}$*.*

Theorem 2 (Completeness). *Given a matching problem* $\boldsymbol{t_1};X(\tilde{x};\tilde{y});\boldsymbol{t_1'} \overset{?}{=} \boldsymbol{t_2}$*, if there is a trace* $\boldsymbol{t}$ *such that* $\boldsymbol{t_1};\boldsymbol{t};\boldsymbol{t_1'} = \boldsymbol{t_2}$*, then the matching algorithm will report the solution* $X \leftarrow \rho\boldsymbol{t}$ *for some renaming* ρ*.*

Because this algorithm embeds two trace equality subproblems, it is at least GI-complete. It may report multiple solutions.

5 Generalizations

In this paper, we have investigated the matching problem for the notion of concurrent traces emerging from multiset rewriting systems (or equivalently place/transition Petri nets [3]). In recent years, more expressive languages for describing concurrent computations have been successfully proposed. State elements can carry information to allow agents to store and exchange data. Some may be read repeatedly rather than consumed on access. In process algebras, synchronization changes the involved processes while our rules are immutable. Each of these extensions, and others, yields much more sophisticated notions of trace. The logical framework CLF [2] provides mechanisms rooted in linear type theory to support these forms of concurrency, and others. Being a type theory, its term language contains constructs that describe the associated traces.

We are extending the matching algorithm outlined here to a large sublanguage of CLF. Supporting reusable state elements, in particular, adds substantial complications. We have also started examining the general matching problem as well as unification for concurrent traces.

References

1. F. Baader and W. Snyder. *Handbook of Automated Reasoning*, chapter Unification Theory, pages 441–526. Elsevier, 2001.

2. I. Cervesato, F. Pfenning, D. Walker, and K. Watkins. A concurrent logical framework II: Examples and applications. Technical Report CMU-CS-02-102, Computer Science Department, Carnegie Mellon University, 2002.
3. I. Cervesato and A. Scedrov. Relating State-Based and Process-Based Concurrency through Linear Logic. *Information & Computation*, 207(10):1044–1077, 2009.
4. V. Diekert and G. Rozenberg, editors. *The Book of Traces.* World Scientific, 1995.
5. Carl Adam Petri. Fundamentals of a theory of asynchronous information flow. In *Proc. IFIP*, pages 386–390, Amsterdam, 1963. North Holland Publ. Comp.
6. V.N. Zemlyachenko, N.M. Korneenko, and R.I. Tyshkevich. Graph isomorphism problem. *Journal of Mathematical Sciences*, 29(4):1426–1481, 1985.