A solution of an interleaving decision problem by a partial order technique

Albert R. Meyer* Alexander Rabinovich†

1 Introduction

1.1 Interleaving versus partial order semantics

Approaches to the semantics of concurrent systems may be divided into two main groups: interleaving and partial order. In the interleaving approach, only the temporal behavior of the events of a run is observable; in the partial order approach, 'causal dependency' between events are considered.

The supporters of the interleaving approach argue that

1. Specifications of concurrent systems always refer only to the temporal behavior and ignore causal behavior.

2. Interleaving semantics are technically much simpler than partial order semantics.

Supporters of the partial order approach argue that this approach gives a better account of the activity of a concurrent system. However, in view of (1), it is difficult to convince a researcher of interleaving semantics that casual aspects are important.

Another argument in favor of partial order semantics appeals to partial order heuristics for verification of interleaving behavior. Recently a number of such heuristics were suggested and in several case studies it was empirically demonstrated that these heuristics were efficient (see recent Proceedings of CONCUR and CAV). However, the partial order heuristics do not improve the complexity of verification.

In our paper another argument in favor of partial order semantics is provided. We consider a decision problem which is formulated in terms of interleaving semantics. The decision algorithm will be given in interleaving terms. However, we developed and proved the correctness of the algorithm by appealing to a partial order semantics.

*MIT Laboratory for Computer Science Cambridge, MA 02139, USA

†Department of Computer Science, The Sackler School of Exact Sciences, Tel Aviv University, Israel 69978

This situation is similar with a situation which often occur in mathematics. For example, to find real valued functions that solve a linear differential equation we solve it over the complex numbers. Similarly, if one believes that only interleaving behavior is real he may gain by considering casual semantics.

1.2 Summary of our results

In this paper we consider the following
Decision problem: Given expressions E_1 and E_2 constructed from variables by the regular operations and shuffle. Is identity $E_1 = E_2$ true for all instantiation of its variables by formal languages?

For example, the identity $(X^*Y^*)^* = (X+Y)^*$ is true because for all languages L_1 and L_2, the languages $(L_1^*L_2^*)^*$ and $(L_1+L_2)^*$ are the same.

The above identity contains only regular operations: concatenation, union and iteration. An easy 'folk' theorem [3] shows that the validity of an identity over regular operations can be verified by instantiating the language variables as single letters. For example, in order to check the validity of $(X^*Y^*)^* = (X+Y)^*$ we instantiate the variables X and Y by a single letters a and b and verify that $(a^*b^*)^* = (a+b)^*$. Checking this variable-free identity is a routine matter of checking equivalence of finite state automata.

In concurrency a very important role is played by parallel composition operators. The simplest of these operator is non-communicating parallel connective $||$, corresponding to shuffle of languages. The above folk theorem fails for the expressions containing shuffle. For, example for single letters a and b, the languages $a||b$ and $ab+ba$ are the same. However, the identity $X||Y = XY+YX$ is not true (indeed, instantiate X by a and Y by bc).

An algorithm for the valid identity problem is provided in this paper. In order to check the validity of an identity $E_1(X_1,\ldots,X_k) = E_2(X_1,\ldots,X_k)$ we will specify (see Theorem 3) finite languages $L_1,\ldots,L_k$ (the languages depends on E_1 and E_2) such that the identity is valid iff the variable-free identity obtained through instantiation of $X_1,\ldots,X_k$ by $L_1,\ldots,L_k$ is true. Checking this last variable-free identity is reduced to the checking of language equivalence of finite state automata.

2 Shuffle Regular Expressions

We presuppose two fixed infinite sets

$$\begin{array}{rll} Act = \{a,\ a_1,\ldots b,\ b_1,\ldots\} & \text{the actions} \\ Var = \{X,\ X_1,\ldots Y,\ Y_1,\ldots\} & \text{the variable symbols.} \end{array}$$

Shuffle regular expressions are defined by the following grammar:

$E ::= x \mid c \mid E+E \mid E;E \mid E||E \mid E^*$, where x ranges over alphabet Var of variable symbols and, c ranges over alphabet Act of constant symbols.

$[\![a]\!]\sigma = \{a\}$
$[\![x]\!]\sigma = \sigma(x)$
$[\![E_1 + E_2]\!]\sigma =$ Union of $[\![E_1]\!]\sigma$ and $[\![E_2]\!]\sigma^{pom}$
$[\![E_1; E_2]\!]\sigma =$ Concatenation of $[\![E_1]\!]\sigma$ and $[\![E_2]\!]\sigma^{pom}$
$[\![E_1 || E_2]\!]\sigma =$ Shuffle of $[\![E_1]\!]\sigma$ and $[\![E_2]\!]\sigma$
$[\![E_1^*]\!]\sigma = ([\![E]\!]\sigma)^*$

Figure 1: Definition of $[\![E]\!]\sigma$

We denote by $FVar(E)$ the set of variables which occur in E.

We say that E is a variable free expression if $FVar(E) = \emptyset$.

We use notation $E\{E_1/X_1 \ldots E_n/X_n\}$ for the expression obtained from E by simultaneous substitution of E_i for X_i. We use $\sum_{i=1}^{n} E_i$ as an abbreviation for for $E_1 + E_2 + \ldots + E_n$.

A string is a finite sequence of actions; we use w, u to range over strings. A string language is a set of strings; we use L to range over string languages.

The operations sum, concatenation, iteration and shuffle are defined in a standard way on the string languages.

We recall that a string w belongs to the shuffle of languages L_1 and L_2 if $w = w_1 u_1 v_2 u_2 \ldots w_k u_k$ where $w_1 w_2 \ldots w_k \in L_1$ and $u_1 u_2 \ldots u_k \in L_2$.

A string language environment for $\{X_1 \ldots X_n\}$ is a function which assigns to X_i a string language. For an expression E and a string language environment σ for a set that contains the free variables of E, the string language $[\![E]\!]\sigma$ is assigned in a standard way by structural induction on the expressions (see Fig. 1).

It is clear that if $\sigma(x) = \sigma'(x)$ for every $x \in Fvar(E)$ then $[\![E]\!]\sigma = [\![E]\!]\sigma'$

3 The Valid Identity Problem

We will consider the following decision problem
Valid Identity Problem:
Input: A pair of shuffle regular expressions E_1 and E_2.
Question: Is the identity $E_1 = E_2$ valid, i.e., are the languages $[\![E_1]\!]\sigma$ and $[\![E_2]\!]\sigma$ equal for every string language environment σ for $Fvar(E_1) \cup FVar(E_2)$.

The main technical result of our paper is

Theorem 1 *The valid identity problem for shuffle regular expressions is decidable.*

Theorem 1 follows from the next two theorems.

Theorem 2 *The valid identity problem for variable free shuffle regular expressions is decidable.*

$sn(X) = sn(a) = 0$
$sn(E_1 + E_2) = sn(E_1; E_2) = max(sn(E_1), sn(E_2))$
$sn(E_1 || E_2) = sn(E_1) + sn(E_2) + 1$
$sn(E^*) = sn(E)$

Figure 2: Shuffle nesting of Expressions

Proof: The problem is easily reduced to the problem of equivalence of finite state automata. □

Notation: The shuffle nesting of an expression E is denoted by $sn(E)$ and is defined in Fig. 2.

Theorem 3 *Let E_1 and E_2 be shuffle expressions over variables $X_1, \ldots, X_n$ such that the shuffle nesting of E_1 and E_2 is bounded by k.*
Let $\{a_{i,j},\ \overline{a}_{i,j}\ :\ i = 1, \ldots, n;\ j = 1, \ldots, k\}$ be distinct actions which do not occur in the expressions $E_1,\ E_2$. Let $SPLIT_k^i$ be the expression $\sum_{j=1}^{k} a_{i,j}; \overline{a}_{i,j}$. Identity $E_1 = E_2$ is valid if and only if the variable free identity
$E_1\{SPLIT_k^1/X_1 \ldots SPLIT_k^n/X_n\} = E_2\{SPLIT_k^1/X_1 \ldots SPLIT_k^n/X_n\}$ is valid.

Proof: In order to proof this theorem we appeal to the notions which were developed in the casual approach to concurrency. Theorem 3 follows from Theorem 7 and Theorem 9, part 2, below. □

4 Pomsets

Definition 1 *(Pratt [6]) A concrete pomset P over set Σ of labels consists of a set of events $Events_P$ which are partially ordered by a relation $\leq_P$ and a function lab_P from $Events_P$ into Σ. A function f is an isomorphism between concrete pomsets P_1 and P_2 if it is label preserving isomorphism between the partial orders of P_1 and P_2. An (abstract) pomset is an isomorphism class of concrete pomsets.*

Throughout the paper we provide some definitions and constructions for concrete pomsets. All these definition/constructions are extended in a natural way to the abstract pomsets.

Definition 2 *Events e_1 and e_2 of a pomset P are* **concurrent** *(notation $e_1\ co_P\ e_2$) if neither $e_1 \leq_P e_2$ nor $e_2 \leq_P e_1$.*

Definition 3 *The* **width** *of a pomset P is the maximal number of mutually concurrent events in the P.*

Definition 4 *A* **pomset language** *over Σ is a set of pomsets over Σ. We say that a pomset language PL has width at most n if all pomsets in PL have width less or equal than n.*

Definition 5 *A concrete pomset P is an* **augmentation** *of a concrete pomset Q if $Events_P = Events_Q$, $lab_P = lab_Q$ and $e_1 \leq_Q e_2$ implies $e_1 \leq_P e_2$ for all $e_1, e_2 \in Events_P$.*

Definition 6 *A concrete pomset P is a* **linearly ordered pomset** *if $\leq_P$ is a linear order over $Events_P$.*

We will identify a linearly ordered pomset over a label set Σ with the corresponding string over alphabet Σ. Also every string language is considered as a pomset language.

Definition 7 *The* **linearization** *of a pomset language PL (notation $Lin(PL)$) is the string language L such that $w \in L$ iff w is a linearly ordered augmentation of a pomset $P \in PL$.*

Notations: A pomset containing only one event labeled by l will be denoted by l. The pomset language containing only one pomset P will be denoted by $\{P\}$; in particular, the language containing only the one element pomset labeled by l will be denoted by $\{l\}$.

5 Refinement

Let P be a pomset and f be a function which assigns a pomset to every event of P.

The f-**expansion** of P is a pomset Q obtained by replacing every event of P by its image. Formally,

$Events_Q = \{(e, e') \ : e \in P, \ e' \in f(e)\}$;

$(e_1, e_2) \leq_Q (e_3, e_4)$ if either $e_1 \leq_P e_3$ or $e_1 = e_3$ and $e_2 \leq_{f(e_1)} e_4$.

$lab_Q((e, e')) = lab_{f(e)}(e')$.

We use the notation $Expan(P, f)$ for the f expansion of pomset P.

Definition 8 *A pomset language environment for a set of labels Σ is a function which assigns a pomset language to every label in Σ.*

Notations We use the notation $[l_1 \rightarrow PL_1, \ l_2 \rightarrow PL_2, \ldots, l_n \rightarrow PL_n]$ for the pomset environment which maps l_i to PL_i, $i = 1, \ldots, n$. We denote by $PLE(\Sigma)$ the set of pomset language environment for Σ. We use α, β to range over pomset language environments. We denote by $SLE(\Sigma)$ the set of string language environments for Σ. We use σ, τ to range over string language environments.

Definition 9 *Let P be a pomset. Let f be a function which assigns a pomset to every event of P and let α be a pomset language environment for Σ. The function f is* **consistent** *with α if for every event e*

1. $f(e) \in \alpha(lab_P(e))$ *if* $lab_p(e) \in \Sigma$,
2. $f(e) = \{lab_P(e)\}$ *otherwise.*

Definition 10 *Let* α *be a pomset language environment for* Σ. *The* α**-refinement** $REF(PL, \alpha)$ *of a pomset language* PL *is* $\{Expan(P, f) \ : \ P \in PL$ *and* f *is consistent with* $\alpha\}$.

The refinement operation has properties similar to substitution operation:

Lemma 4 *Let* PL *be a pomset language over an alphabet* Σ *and* α *be a pomset language environment for an alphabet* Σ'. *If* $\Sigma \cap \Sigma' \subseteq \{l_1, \ldots, l_k\}$ *then*
$REF((REF(PL, [l_1 \to PL_1, \ldots, l_k \to PL_k]), \alpha) =$
$= REF(PL, [l_1 \to REF(PL_1, \alpha), \ldots, l_k \to REF(PL_k, \alpha)])$.

The next lemma state how linearization operator interacts with refinement.

Lemma 5 $Lin(REF(PL, [l_1 \to PL_1, \ldots, l_k \to PL_k]) = Lin(REF(PL, [l_1 \to Lin(PL_1), \ldots, l_k \to Lin(PL_k)]$

6 Operations definable by pomset languages

Definition 11 *The* **application of a pomset language** PL *to a string language environment* τ *(notation* $PL \bullet \tau$*) is the string language defined as* $Lin(REF(PL, \tau))$.

Definition 12 *Let* F *be a function from string language environments for* Σ *into string languages. We say that* F *is* **definable by a pomset language** PL *if* $F\tau = PL \bullet \tau$ *for all* $\tau \in SLE(\Sigma)$.

Observation 6 *The regular operations and shuffle are pomset language definable, namely*

1. *Let* $PAR(X, Y)$ *be the pomset consisting of two unordered events labeled by* X *and* Y. *It is easy to see that* $[\![X||Y]\!]$ *defines the same operation as* $\{PAR(X, Y)\}$ *over the string environments for* $\{X, Y\}$.
2. *Let* $SEQ(X, Y)$ *be a pomset with two events labeled by* X *and* Y *such that the event* X *precedes the event* Y. *It is easy to see that* $[\![X; Y]\!]$ *defines the same operation as* $\{SEQ(X, Y)\}$ *over the string environments for* $\{X, Y\}$.
3. *Let* $SUM(X, Y)$ *be the pomset language consisting of two one element pomsets* X *and* Y. *It is easy to see that* $[\![X + Y]\!]$ *defines the same operation as* $SUM(X, Y)$ *over the string environments for* $\{X, Y\}$.
4. *Let* $ITER(X)$ *be the pomset language which consists of all finite strings over symbol* X. *It is easy to see that* $[\![X^*]\!]$ *defines the same operation as* $ITER(X)$ *over the string environments for* $\{X\}$.

$[\![a]\!]^{pom} = \{a\}$
$[\![x]\!]^{pom} = \{x\}$
$[\![E_1 + E_2]\!]^{pom} = REF(SUM(X,Y),\ [X \to [\![E_1]\!]^{pom},\ Y \to [\![E_2]\!]^{pom}])$
$[\![E_1; E_2]\!]^{pom} = REF(SEQ(X,Y),\ [X \to [\![E_1]\!]^{pom},\ Y \to [\![E_2]\!]^{pom}])$
$[\![E_1 || E_2]\!]^{pom} = REF(PAR(X,Y),\ [X \to [\![E_1]\!]^{pom},\ Y \to [\![E_2]\!]^{pom}])$
$[\![E_1^*]\!]^{pom} = REF(ITER(X),\ [X \to [\![E]\!]^{pom}])$

Figure 3: Pomset Semantics

Theorem 7 *For every shuffle regular expression E, the operation $\lambda\sigma.\ [\![E]\!]\sigma$ is definable by a pomset language with width bounded by the shuffle nesting of E.*

Proof: By structural induction on the expressions the pomset language $[\![E]\!]^{pom}$ is assigned to every shuffle regular expression (see Fig. 3). Relying on Lemma 4, Lemma 5 and Observation 6, it can be shown that $[\![E]\!]\sigma = [\![E]\!]^{pom} \bullet \sigma$, where σ is any language environment σ for the $Fvar(E)$. □

Definition 13 *A string language environment $[l_1 \to L_1, \dots l_n \to L_n]$, is called a split-choice environment for $\{l_1 \dots l_n\}$ if every L_i contains only strings of length two.*

Lemma 8 *$PL \bullet \tau = PL' \bullet \tau$ for every string language environment τ for Σ iff $PL \bullet \tau = PL' \bullet \tau$ for every split-choice environment τ for Σ.*

This lemma can be strengthened as follows:

Theorem 9 *Let PL and PL' be pomset languages over an alphabet Σ and let $\{a_{i,j},\ \overline{a}_{i,j}\ :\ i = 1, \dots, n\ ,\ j \in Nat\}$ be distinct labels not in Σ. Let $L_i^{(\infty)}$ be string language $\{a_{i,j}; \overline{a}_{i,j}\ \ : j \in Nat\}$ and let $L_i^{(k)}$ be string language $\{a_{i,j}; \overline{a}_{i,j}\ \ : j = 1, 2, \dots, k\}$.*

1. *$PL \bullet \tau = PL' \bullet \tau$ for every string language environment τ for $\{m_1, \dots, m_n\}$ iff $PL \bullet [m_1 \to L_1^{(\infty)}, \dots, m_n \to L_n^{(\infty)}] = PL' \bullet [m_1 \to L_1^{(\infty)}, \dots, m_n \to L_n^{(\infty)}]$.*
2. *Let PL and PL' be pomset languages of width at most k. Then $PL \bullet \tau = PL' \bullet \tau$ for every string language environment τ for $\{m_1, \dots, m_n\}$ if and only if $PL \bullet [m_1 \to L_1^{(k)}, \dots, m_n \to L_n^{(k)}] = PL' \bullet [m_1 \to L_1^{(k)}, \dots, m_n \to L_n^{(k)}]$.*

Remarks (1) The above theorem can be strengthened as follows: Let k_i be the bound on the number of mutual concurrent events labeled by m_i in the pomset languages PL and PL'. Then $PL \bullet \tau = PL' \bullet \tau$ for every string language environment τ for $\{m_1, \dots, m_n\}$ if and only if
$PL \bullet [m_1 \to L_1^{(k_1)}, \dots, m_n \to L_n^{(k_n)}] = PL' \bullet [m_1 \to L_1^{(k_1)}, \dots, m_n \to L_n^{(k_n)}]$.
(2) Weaker versions of the above theorem have appeared in the literature.

Gischer [1] considered the operations definable by pomsets. One of his results can be stated as follows: For pomsets P and P' with less than k events, $\{P\} \bullet \tau = \{P'\} \bullet \tau$ for every string language environment τ for $\{m_1, \ldots, m_n\}$ if and only if $\{P\} \bullet [m_1 \to L_1^{(k)}, \ldots, m_n \to L_n^{(k)}] = \{P'\} \bullet [m_1 \to L_1^{(k)}, \ldots, m_n \to L_n^{(k)}]$.

In [5] the special pomsets which are called semi-words are considered. A pomset is a semi-word if no events with the same label are concurrent. It was proved in [5] that for semi-word languages SWL and SWL' the following theorem holds: $SWL \bullet \tau = SWL' \bullet \tau$ for for every string language environment τ for $\{m_1, \ldots, m_n\}$ if and only if
$SWL \bullet [m_1 \to L_1^{(1)}, \ldots, m_n \to L_n^{(1)}] = SWL' \bullet [m_1 \to L_1^{(1)}, \ldots, m_n \to L_n^{(1)}]$.

(3) The proof of Theorem 9 can be extracted from the proofs of these two weaker versions.

7 Further Results

7.1 Complexity of the valid identity problem

An exponential space algorithm can be provided for the valid identity problem. Mayer and Stockmeyer [2] provided EXPSPACE lower bound for the valid identity problem of the variable free shuffle regular expression. These results give a tight lower and upper bound for the valid identity problem.

7.2 Extension by other pomset language definable operations

Let OP be an n-ary operation on string languages. We say that OP is effective on regular languages if there exists an algorithm which constructs a finite automaton for the language $OP(L_1, \ldots, L_n)$ from finite automata for $L_1, \ldots, L_n$.

We say that OP is definable by (finite width) pomset language if there exists a (finite width) pomset language PL such that $PL \bullet [1 \to L_1, \ldots, n \to L_n] = OP(L_1, \ldots, L_n)$ for any languages $L_1, \ldots, L_n$.

Note that the operations definable by finite pomset languages are effective on regular languages. Among such operations are operations which are not definable by any shuffle regular expressions.

The valid identity problem is decidable for the expressions constructed over any set of operations which are effective on regular languages and are definable by finite width pomset languages.

7.3 Extension by Intersection

Micciancio [4] proved the decidability of the valid identity problem for constant free shuffle-intersection regular expressions. These expressions are defined by the following grammar: $E ::= \ x \mid E \cap E \mid E + E \mid E; E \mid E || E \mid E^*$, where x ranges over variable symbols.

Note that the intersection is not a pomset language definable operation. Micciancio's very interesting proof is given in terms of interleaving semantics and does not use explicitly pomsets. It is an open problem whether his results and techniques can be extended to other pomset language definable operations and in particular to the expressions which contain constants.

Acknowledgments

We would like to thank Yoram Hirshfeld and Boris Trakhtenbrot for stimulating discussions and many constructive suggestions.

References

[1] J. Gischer. Partial Orders and Theory of Shuffle. PhD thesis, Stanford University, 1984.

[2] A. Mayer and L. Stockmeyer. The complexity of word problem - this time with interleaving. Information and Computation, pages 293-311, Vol 115, 1994.

[3] A. R. Meyer. Concurrent Process Equivalences: Some decision problems. In STAC 95, volume 900 of *Lect. Notes in Computer Science*, Springer Verlag, 1995.

[4] D. Micciancio. The Validity problem for Extended regular Expressions. M SC thesis, MIT, 1996.

[5] M.. Nielsen, U. Engberg, and K. Larsen. A Simple Process Language with Refinement. In *REX Workshop on Linear Time, Branching time and Partial Order in Logics and Models for Concurrency* , volume 354 of *Lect. Notes in Computer Science*, pages 523-548, Springer Verlag, 1990.

[6] V. R. Pratt, On the composition of Processes. In *Proc. of the Ninth Annual ACM Symposium on Principles of Programming Languages*, 1982.