Trace Theory and State Explosion

M. W. Mislove
Department of Mathematics
Tulane University
New Orleans, LA, U.S.A.

Abstract *Trace theory is a method for modeling concurrency in which concurrent computation is supported explicitly, rather than relying on sequential composition, nondeterminism and interleaving. The state explosion problem arises in model-checking because of the plethora of states that can arise in the interleaving approach to modeling even simple algorithms. In this paper we explore the relationship between a new approach to using trace theory to model concurrent computation and the state explosion problem. Trace theory already has been recognized as having utility in controlling the state explosion in such examples; our point of departure is to utilize trace theory more directly, rather than relying simply on the basic tenets of that approach.*

Keywords: Trace theory, model checking, software tools, concurrency.

1 Introduction

Trace theory is an approach to modeling concurrent computation which supports concurrent computation directly, rather than starting with sequential composition and using nondeterminism and then interleaving to build a model for parallel composition – and hence concurrency. The latter is the traditional method for modeling concurrency in programming languages; examples abound – see, e.g. [4]. But the main applications of trace theory have been to algorithmic analysis and computational complexity. This is due in part to the fact that the original purpose of trace theory was to provide models for Petri nets, themselves models for nondeterministic automata. But there are fundamental problems with the basic models that trace theory offers that prevent their use as models for programming languages.

Nonetheless, trace theory has had a powerful impact on our understanding of concurrency, and its utility can be found in a wide range of areas. One of these areas is *model checking*, where trace theory has provided what are called "partial order methods." These methods afford partial solutions to the *state explosion problem*. This problem arises because of the exponential increase in the number of states that can arise in a concurrent system as the number of states of the component processes increases. This exponential increase prevents exhaustive search from being an effective method to verify that such a system meets its specification, and so alternative methods are needed for validation. One of these approaches is partial order methods, whose application relies on the fact that it often happens that not all states need to be validated in order to validate the system. This observation is due to the fact that often a number of distinct computations all lead to the same output for a given input. In such a situation one needn't explore all these different computation paths – exploring one should suffice. The problem then comes down to the following issues:

- Find which paths have the same overall effect on the system.
- For each set of paths having the same effect, find a representative path to validate.

- Find effective methods to carry out the previous two steps.

Recently [2] trace-theoretic models have been devised which have the structure of a *domain* in which the concatenation operation of trace theory is continuous. Such structures have long been used to provide models for programming languages, primarily because of the ease with which recursion can be modeled in them. In [1] a simple concurrent programming language is presented which uses the concatenation operator of trace theory as its primary operation, and the resource traces model of [2] is used as the basis for a denotational model for this language. The main result of [1] is a congruence theorem between an operational model for the language under study and the denotational model built using the resource traces model, thus allowing for the first time a denotational semantics for true concurrency that supports recursion in the underlying language.

The thrust of this paper is to explore the utility of the language and its models presented in [1] as a methodology for applying trace theory in model checking. We focus on the work in the seminal thesis [3] as our main source for partial order methods in model checking, and we show how the language from [1] can be used to address some of the issues raised in that work. In particular, the need for a flexible modeling system that emerges in [3] seems in part to be satisfied by the results in [1].

The rest of the paper is organized as follows. In the next section we present a brief summary of the approach to model checking that is the focus of [3]. Then, in the following section, we present a brief outline of the approach to concurrency that is taken in [1]. The next section the explores the use of the work in [1] as a method for modeling the concurrent systems studied in [3]; in particular, we explore the potential that the models of [1] have for studying the issues raised in [3]. While our results are very preliminary, we believe they show a promising avenue for achieving the needs outlined in the list of goals given above.

2 Partial order methods for model checking

In this section we present a brief synopsis of the application of partial order methods to model checking. Our presentation focuses on the work in [3], which is a survey of results along this line. We find this a particularly appropriate presentation for our purposes, since one of the central issues in [3] is the myriad relations which arise in the application of partial order methods to distinct problems.

The approach taken in [3] views a concurrent system as a quadruple $(\mathcal{P}, \mathcal{O}, \mathcal{T}, s_0)$, where

$\mathcal{P}$ is a finite set of *processes*, $P_1, \ldots, P_m$ which are pairwise disjoint, each with local states, and each of which can operate on the objects in $\mathcal{O}$.

$\mathcal{O}$ is a finite set of objects, $O_1, \ldots, O_n$, each of which consists of a pair (V_i, OP_i), where V_i is the set of values that are possible for the object, and OP_i is the set of operations that can be applied to the object.

$\mathcal{T}$ is a finite set of transitions of the system.

$s_0 \in S = P_1 \times \cdots P_m \times V_1 \times \cdots V_n$ is the initial state of the system. In general, *global states* consist of a local state $s(i)$ for each process P_i, and a value $v_i \in V_i$ for each object O_i.

Processes act not only on their own local states, but also on the objects in $\mathcal{O}$. This latter is done by taking an input IN_i to process P_i together with a value v_j of object O_j and associating an output in OUT_i and a corresponding value $v_i' \in V_i$.

Transitions in this system are quadruples, $t = (L, G, C, L')$ consisting of *partial control states* $L, L' \subseteq \cup_{i \leq m} P_i$. The assumption is that $L \cap P_i$ and $L' \cap P_i$ contain at most one element, and $L \cap P_i \neq \emptyset$ iff $L' \cap P_i \not\emptyset$; i.e., each local state has at most component from any of the P_i, and both L and L' have local states in P_i iff either one does. This enforces a component-like action of transitions on processes. The second component G of a transition is a *guard*, which

consists of a conjunction of conditions each of which can test the current value of objects, but cannot change their values. The third component is the *command* C which is a self-map of $V_1 \times \cdots \times V_n$. This command consists of the sequential composition of operations on objects and must satisfy the condition that, once one of these operations has modified an object O_i, then no succeeding element of the command C can operate on O_i.

The transition t is *enabled* in a state s iff $L \subseteq s$ and G is true in s. From this it follows that L has at most value in each process P_i, although there may be many states in which t is enabled. It also follows that transitions are deterministic because execution of a transition leads to a unique successor state, L'.

In [3] a semantics is presented for the systems we have just described. Using a labeling function $\nu: \mathcal{T} \to \Sigma$ and the resulting automaton $A_G = (\Sigma, S, \Delta, s_0)$ whose actions come from the alphabet Σ and whose transition relation is

$$\Delta \equiv \{(s, a, s') \mid (\exists t \in \mathcal{T})\ s \xrightarrow{t} s' \wedge \nu(t) = a\}.$$

The interest then becomes the states of A_G that are reachable from s_0. Since the system is assumed finite, A_G has only finitely many states, and so, in principle, one could explore all the states of the system. Unfortunately, this is not a practical approach to analyzing the system – as the number of states of each component grows, the number of states of the system grows exponentially (according to the possible interleavings of the possible transitions). Thus methods are sought to perform an analysis of the system without exploring all states.

One approach to such analyses is based on the observation that if two transitions t and t' are *independent* – e.g., if neither affects the states of the other – then $tt' = t't$ in terms of the result of applying both states in turn. Clearly one need not explore both tt' and $t't$ in this case – exploring one will do. *Partial order methods* can be viewed as an attempt to be more precise about which transitions can be regarded as independent, with the aim of doing a *selective search* of the state space in which only one execution path is explored for each family of transitions that are mutually independent. It is interesting to note that there does not seem to be one fixed notion of independence that is applicable for all applications, and so part of the problem is to find notions of independence that lead to sets of transitions that:

- result in transitions whose order of computation is irrelevant in terms of the output of the computation, and
- which provide relations on the set of transition that provide tractable equivalence classes of independent transitions. I.e., there are efficient algorithms for computing the sets of transitions that become identified as being independent of one another.

In [3], the "underlying" notion of dependency (the opposite of independence) can be stated as follows:

Definition 3.1: A binary relation $D \subseteq \mathcal{T} \times \mathcal{T}$ is a *valid dependency relation* iff $(\forall t, t' \in \mathcal{T})$ $(t, t') \notin D$ implies $\forall s \in S$

1. If t is enabled in state $s \in S$ and $s \xrightarrow{t} s'$, then t' is enabled in s iff t' is enabled in s'.
2. If t and t' are both enabled in s, then there is a unique state s' such that $s \xrightarrow{tt'} s'$ and $s \xrightarrow{t't} s'$.

The problem is that these conditions for a valid dependency relation are difficult to verify, and so a great deal of the discussion in [3] focuses on sufficient conditions for a valid dependency relation which are practical to validate for transitions of the system. As we shall see in the next section, some of these conditions also allow the semantic models developed in [1] to be used to model the system under study.

3 A model for true concurrency

In this section we outline the results from [1] which present a simple concurrent programming language and give both an operational and a denotational model for the language. In addition, it is shown that these two semantics are equivalent, in the sense that the behavior that the operational semantics assigns to a term of our language is the same as the meaning of the term in the denotational model (such a result is called a *congruence theorem* because it shows that the behavior mapping the operational mode defines is a congruence with respect to all the operations the language supports).

The approach presented in [1] is based on *resource traces*, developed in [2]. In this approach, one begins with an alphabet Σ of *actions* which processes in the language can execute, together with a *resource mapping* $\text{res}: \Sigma \to \mathcal{P}(\mathcal{R})$ which associates to each action $a \in \Sigma$ a *non-empty* set of resources it needs to complete its execution. One can view these resources as ports, memory, etc. – or even the states of the system that the action needs to read from or write to. From this mapping one determines the *dependency relation* as

$$(a, b) \in D \;\Leftrightarrow\; \text{res}(a) \cap \text{res}(b) \neq \emptyset,$$

so that actions are dependent iff they share some resource. Thus, if we vary the resources $\mathcal{R}$ or the resource mapping res, then we vary which actions are independent and which are dependent.

The language $\mathcal{L}$ studied in [1] has a BNF-like syntax consisting of

$$p ::= \text{STOP} \mid a \mid p \circ p \mid p|_R \mid p \underset{C}{\|} p \mid x \mid \text{rec}\, x.p.$$

Here, STOP is the deadlocked process, $a \in \Sigma$ is the process that executes the action a and normally terminates, and $p \circ q$ is the concurrent computation of the two component processes: in this composition, independent actions are allowed to commute past one another, while dependent actions must occur in the order in which they are written. For a set $R \subseteq \mathcal{R}$ of resources, the process $p|_R$ acts like p, except that all actions *must* have their resources lying inside R. If $C \subseteq \mathcal{R}$ is a set of resources, then we can regard them as channels over which distinct processes can synchronize, and then $p \underset{C}{\|} q$ is the process which forces the components to synchronize on the channels in C, and that perform actions whose resources do not intersect C or the resources of the other component independently. If these conditions are violated, then the synchronized product deadlocks. Finally, $x \in X$ is a process variable, and $\text{rec}\, x.p$ denotes recursion in the variable x.

3.1 A denotational semantics for $\mathcal{L}$

An easily defined denotational model for the language $\mathcal{L}$ is presented in [1] uses the resource mapping and is based on the resource traces model of [2]. The approach taken in [1] is to extend the resource mapping $\text{res}: \Sigma \to \mathcal{P}(\mathcal{R})$ to a mapping $\text{res}: \mathcal{L} \to \mathcal{P}(\mathcal{R})$ which assigns to each process its set of resources. One first defines this mapping for *finitary terms* – those without process variables. The easiest way to do this is to define an interpretation of each of the operators from the BNF for $\mathcal{L}$ on $\mathcal{P}(\mathcal{R})$. For example, we can interpret STOP as $\mathcal{R}$, each action $s \in \Sigma$ as $\text{res}(a)$, each of $\circ$ and $\underset{C}{\|}$ as union, and restriction to the subset $R \subseteq \mathcal{R}$ as intersection with R. This makes $\mathcal{P}(\mathcal{R})$ into an algebra with the same signature as the finitary portion of $\mathcal{L}$. Of course, our language also has process variables, and so we cannot properly define the extension of res to all of $\mathcal{L}$ without first assigning resources to each variable x. Using the set of semantic resources, $\mathcal{P}(\mathcal{R})^X$ of such assignments, the algebra structure we have just defined on $\mathcal{P}(\mathcal{R})$ extends to the set of continuous maps $[\mathcal{P}(\mathcal{R})^X \to \mathcal{P}(\mathcal{R})]$, and this also has an interpretation of variables $x \in X$ and of recursion, this last using least fixed points (with the usual set containment order on $\mathcal{P}(\mathcal{R})$). Then universal algebra ensures there is an algebra homomorphism from $\mathcal{L}$ to $[\mathcal{P}(\mathcal{R})^X \to \mathcal{P}(\mathcal{R})]$ since $\mathcal{L}$ is the initial algebra of this signature.

The definitive denotational model for $\mathcal{L}$ is more complicated. It relies on the notion of a resource trace as defined in [2]. Briefly, a *trace* over the alphabet Σ is an isomorphism class of a labeled graph (V, E, λ) where V is a countable set of vertices, $E \subseteq V \times V$ is a set of edges, and $\lambda: V \to \Sigma$ is a labeling which satisfies the property that the pair $(v, v') \in E$ iff $\lambda(v)$ and $\lambda(v')$ are dependent in Σ or are equal. The set of equivalence classes of traces with finite vertex sets is isomorphic to the trace monoid obtained by considering the quotient space $\Sigma^*/\equiv$ of finite words over Σ modulo the congruence $\equiv$ (with respect to concatenation of words) generated by the family $\{(ab, ba) \mid (a, b) \in I\}$, where I is the set of pairs of independent actions. The problem is that, while the concatenation operation for words induces a monoid operation of concatenation of traces, this operation is not monotone with respect to the prefix order relation given by: $p \leq q$ iff $(\exists r)\ q = p \circ r$. This is a partial order on the set of traces, and it is left cancellative; i.e, the trace r for which $q = p \circ r$ is unique (just as for words).

This defect of traces was corrected in [2] with the notion of a resource trace. This begins with the notion of the *resources at infinity* of a trace, which are defined as $\text{resinf}(p) = \cap\{\text{res}(k^{-1}p) \mid k \leq p \ \wedge k \text{ is finite}\}$. Gastin and Teodosiu then define the set of resource traces as the family

$$\mathbb{F} = \{(r, R) \mid r \text{ is a trace } \wedge \text{ resinf(r)} \subseteq \text{R}\}.$$

Using the same techniques as described above, but with this richer notion of concatenation of resource traces, we can make $\mathbb{F}$ and then $[\mathbb{F}^X \to \mathbb{F}]$ into an algebra with the signature of $\mathcal{L}$, and then again deduce the existence of an algebra homomorphism $\mathcal{M}: \mathcal{L} \to [\mathbb{F}^X \to \mathbb{F}]$. The interesting aspect of the denotational semantics is that the meaning of a recursive term is *not* given as a least fixed point. Indeed, the least element of $\mathbb{F}$ is the pair $(1, \mathcal{R})$, which claims all resources, and so using this as the starting point for iterating a self-map would prevent any other action from occurring. This problem is remedied by first computing for a recursive term $\text{rec}\, x.p$ the set of resources R needed only for it to complete its execution, and then iterating the self-map that is the meaning of p starting at the pair $(1, R)$. This means that only those actions which actually require resources that p needs are blocked, and all other actions can commute with this recursive term.

Table 1: Some Transition Rules for $\mathcal{L}$

$$\frac{}{a \xrightarrow[\sigma]{a} \text{SKIP}}$$

$$\frac{p \xrightarrow[\sigma]{a} p'}{p \circ q \xrightarrow[\sigma]{a} p' \circ q}$$

$$\frac{q \xrightarrow[\sigma]{a} q',\ \text{res}(a) \cap \text{res}(p, \sigma) = \emptyset}{p \circ q \xrightarrow[\sigma]{a} p \circ q'}$$

$$\frac{p \xrightarrow[\sigma']{a} p',\ \text{where } \sigma' = \sigma[x \mapsto \text{res}(\text{rec } x.p, \sigma)]}{\text{rec } x.p \xrightarrow[\sigma]{a} p'[\text{rec } x.p/x]}$$

3.2 An operational semantics for $\mathcal{L}$

An operational semantics for $\mathcal{L}$ also is presented in [1]. This semantics is derived from a set of transition rules which indicate which actions a process in the language can execute. By following all possible paths of execution, one discovers the *behavior* of a process. The complete set of transition rules for $\mathcal{L}$ are presented in [1]; Table 1 lists only those that are relevant to our discussion; here $\text{SKIP} = \text{STOP}|_\emptyset$ denotes normal termination, and $\sigma: X \to \text{L}$ is a syntactic environment. The fact that the behavior of a term – i.e., the maximal trace that it can execute under these rules, and there is one and only one – is the same as the denotational meaning of the term as given by the mapping $\mathcal{M}$ described above means that we can either calculate the behavior using these rules, or we can calculate it using the properties of the mapping $\mathcal{M}$, which is an algebra homomorphism.

4 Relating models and semantics

We now describe some relations between the presentation of partial order methods that we outlined in Section 2 to the semantic considerations of Section 3. The main point is rather simple to state:

> *Models for partial order methods can be captured by the semantic models for $\mathcal{L}$.*

Implicit in this statement is the assertion that we can view the trace equivalence class of each word of the automaton A_G as the meaning of a term in the language $\mathcal{L}$. We show how to accomplish this for a specific set of syntactic conditions that define a valid dependency relation in Section 2. Recall that the conditions for a valid dependency relation of Definition 3.1 are hard to validate directly, and so a number of sufficient conditions are studied in [3] which produce valid dependency relations, but which are more tractable. One of those sets of conditions for transitions t and t' to be independent are:

1) The set of processes P_i that are active for t is disjoint from the set of processes for which t' is active, and

2) the set of objects that are accessed by t is disjoint from the set of processes that are accessed by t'.

Let's make the following assumptions about our system.

1. The family of all processes and objects, taken together, are pairwise disjoint.

2. The labeling function $\nu: \mathcal{T} \to \Sigma$ satisfies $\nu(t) = \nu(t')$ iff

   - The processes active for t are the same as the processes active for t', and
   - The objects accessed by t are the same as the objects accessed by t'.

   Of course, if ν is one-to-one, then these conditions are trivially satisfied.

Now, define the set of resources $\mathcal{R} = \mathcal{P} \cup \mathcal{T}$ to be the union of the sets of processes and objects of our system, and define the resource map res: $\Sigma \to \mathcal{R}$ by

$$\begin{aligned} \text{res}(a) &= \{P_i \mid (\exists t \in T_a)\ P_i \text{ is active for t}\} \\ &\cup \{O_j \mid (\exists t \in \mathcal{T}_a)\ O_j \text{ is accessed by t}\}, \end{aligned}$$

where $\mathcal{T}_a = \{t \in \mathcal{T} \mid \nu(t) = a\}$ for each $a \in \Sigma$. The conditions we have imposed on ν guarantee that res is well-defined. Moreover, the following result is clear:

Proposition 4.1: The transitions t and t' are independent as defined by conditions 1) and 2) above iff the associated actions $\nu(t)$ and $\nu(t')$ are independent in the sense that $\text{res}(\nu(at)) \cap \text{res}(\nu(t')) = \emptyset$. □

Thus we can view the transitions of our system as having the same independence properties as the actions $a \in \Sigma$. Moreover, the work cited from [1] implies that we can regard the words of the automation A_G as being generated by terms of the language $\mathcal{L}$, and so the equivalence classes of these words are the same as the equivalence classes of the associated semantic mapping $\mathcal{M}: \mathcal{L} \to [\mathbb{F}^X \to \mathbb{F}]$. Finally, since this mapping is equivalent to the one defined by the behavior mapping the operational semantics defines, words from A_G are equivalent iff they terms which generate them have the same behaviors as given by the transition system for the operational semantics of $\mathcal{L}$. We can summarize all this in the following

Theorem 4.2: Let w and w' be words of the automaton A_G for the concurrent system outlined in Section 2, and let p and p' be any two terms from $\mathcal{L}$ whose meanings are the equivalence class of w and w', respectively. Then w and w' are equivalent in A_G for the independence relation defined by 1) and 2) above iff $\mathcal{M}(p) = \mathcal{M}(p')$, iff p and p' have the same behavior under the operational semantics defined for $\mathcal{L}$. □

5 Summary

We have described the approach to partial order methods outlined in [3] and the denotational and operational semantics developed in [1] for a simple parallel programming language. Using these as basis. we have presented a relation between the two, showing how the semantics of [1] captures the equivalence of computation paths represented by the automaton for the concurrent system being analyzed in [3]. This means that the problem of finding equivalent computation paths can be solved by the problem of deciding which terms of the language studied in [1] have the same meaning. The methods of semantics – both denotational and operational – are thus available to help analyze concurrent systems. In particular, we can utilize the fact that we are dealing with a language Ł which can be viewed as a universal algebra and so the equivalence relation that identifies words with their trace equivalence class is an algebra congruence.

Unfortunately, it is unclear what contribution this provides, other than the obvious one of validating a strong connection between two areas. In fact, we have presented only one example of a independence relation which translates faithfully from the model-checking setting to the semantics setting. Our intention is to investigate further which of the relations studied in [3] as well as in other approaches using partial order methods are amenable to the techniques we have described. Since much effort has been put into finding efficient algorithms for computing selective searches in the model-checking setting, it is apparent that those algorithms may have utility in the semantics community by providing efficient computational strategies for finding the operational behavior of terms from our language $\mathcal{L}$. In the other direction, it is hoped that the modularity with which the denotational meaning of terms in the language can be analyzed will provide some help to the model-checking community in finding more efficient selective search algorithms.

Acknowledgements

The ideas presented here are based in large part on the work reported in [1], which is the result of an on-going collaboration with Professor Paul Gastin, Université de Paris VII. The results presented in [1] rely crucially on the work in [2], and clearly the collaboration that resulted in [1] could not have occurred without the efforts of Professor Gastin. But the ideas expressed here are the author's, so any errors in this presentation are due solely to him.

The author also gratefully acknowledges the partial support of the National Science Foundation and the Office of Naval Research while carrying out the research reported in this paper.

References

[1] Paul Gastin and Michael Mislove. A truly concurrent semantics for a simple programming language. submitted.

[2] Paul Gastin and Dan Teodesiu. Resource traces: a domain for process sharing exclusive resources. *Theoretical Computer Science*, to appear.

[3] Patrice Godefroid. *Parial-order Methods for the Verification of Concurrent Systems.* PhD Thesis. Université de Liege, 1994.

[4] A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall, London, 1998.