Graphical Models of Separation Logic

Ian Wehrman[a], C. A. R. Hoare[b], Peter W. O'Hearn[c]

[a] *The University of Texas at Austin, USA*
[b] *Microsoft Research Cambridge, UK*
[c] *Queen Mary University London, UK*

Abstract

Graphs are used to model control and data flow among events occurring in the execution of a concurrent program. Our treatment of data flow covers both shared storage and external communication. Nevertheless, the laws of Hoare and Jones correctness reasoning remain valid when interpreted in this general model.

Key words: concurrency, formal semantics.

1. Introduction

In this paper, we present a trace semantics based on graphs: nodes represent the events of a program's execution, and edges represent dependencies among the events. The style is reminiscent of partially ordered models [11, 16], though we do not generally require properties like transitivity or acyclicity. A linear trace can be represented by a graph in which there is a chain of arrows between every pair of nodes. But we also allow any node to have mutually independent predecessors on which it depends, and successors which it enables.

Concurrency and sequentiality are defined using variations on separating conjunctions. Whereas the conjunction in the original separation logic partitions addresses in a heap [10, 14], the conjunctions here partition events in a trace. In this way, the model can be thought of as a spatial logic [1, 4], though it lacks such features as a mechanism for hiding names or expressing locality, and does not ensure that satisfying models are closed under isomorphism. Nor does the model make use of an explicit notion of resource, as in [13]. However, the model does have many pleasant algebraic properties, which are shown with surprisingly simple proofs, presented in a companion report [6]. We present a number of theorems about the generic model, including the soundness of the laws of Hoare logic [5] and the Jones rely/guarantee calculus [7]. No particular languages or applications are studied in the paper; we leave this to future work.

2. Traces and Separation

A *directed graph* is a pair of sets (EV, AR), where EV is a set of nodes and AR is a set of directed arrows linking the nodes. We think of the objects of

EV as representing occurrences of atomic events recorded in a trace of program execution, and the objects of AR represent control or data flows that occur between events. If p and q are events, we write $p \rightarrow q$ to indicate the existence of an arrow between them. A *labelled graph* also has a labelling function on its events and arrows. In this paper, we are interested only in event labels: $\mathit{label}(e) = \ell$ means that ℓ is the atomic action of the programming language (e.g., $x := x + 2$) whose execution is recorded as e.

The sets EV and AR are considered to contain all possible events in the execution of a program, so as to constitute a kind of complete trace. From this complete execution, smaller, possibly incomplete traces will be selected by the algebraic operations defined in this paper. These *portions* of the complete execution record the execution of a component of the program, and will themselves be called simply *traces*.

Formally, a trace is a subset of EV. A singleton trace is called *atomic*; we denote by $[\ell]$ the set of singleton traces in which the only event has label ℓ:

$$[\ell] =_{\mathbf{def}} \{tr \mid \exists e \in tr \,.\, tr = \{e\} \ \&\ \mathit{label}(e) = \ell\}.$$

We can define the semantics of a program as the set of its possible traces, as in the CSP traces model [15]. For example, $[\ell]$ is the set of traces associated with a label (thought of as a program) ℓ.

We write $P * Q$ for the structured command that denotes the concurrent execution of components P and Q. No event is simultaneously part of the execution of both these commands, but every event is in the execution of at least one of P and Q. Therefore, the most general form of a trace tr of $P * Q$ is the disjoint union of some trace tp of P with some trace tq of Q:

$$tr = tp * tq \ \equiv_{\mathbf{def}}\ tr = tp \cup tq \ \&\ tp \cap tq = \emptyset.$$

This partial function on traces can be lifted to a total function on sets of traces in the usual way:

$$P * Q \ =_{\mathbf{def}}\ \{tr \mid tr = tp * tq \ \&\ tp \in P \ \&\ tq \in Q\}.$$

We call the $(*)$ function on trace sets the *concurrent separator*. In words, a trace is a model of $P * Q$ exactly when it can be split into two disjoint parts, one of which is a trace of P and the other a trace of Q. This definition can be implemented by running P and Q concurrently, as separate threads or processes in the same or in different computers. There is no restriction on the arrows which communicate between events of P and events of Q. The two threads may communicate freely with each other, e.g. through shared memory or channels.

A stronger notion is *sequential separation*. Informally, a trace tr may be split into a sequential separation $tp\,;tq$ when there is no arrow from an event of tq to an event of tp:

$$tr = tp\,;tq \ \equiv_{\mathbf{def}}\ tr = (tp * tq) \ \&\ \forall p \in tp \,.\, \forall q \in tq \,.\, \neg(q \rightarrow p).$$

Again, this trace separator is lifted pointwise to a total function on trace sets:

$$P\,;Q \ =_{\mathbf{def}}\ \{tr \mid tr = (tp\,;tq) \ \&\ tp \in P \ \&\ tq \in Q\}.$$

This definition expresses an essential property of sequential composition. It allows implementations to optimise a program by interleaving events of the first trace with events of the second; the events can even be executed concurrently, if they do not violate the dependency condition in the definition. We have thus defined what is sometimes called a "weak sequential composition" in concurrency theory.

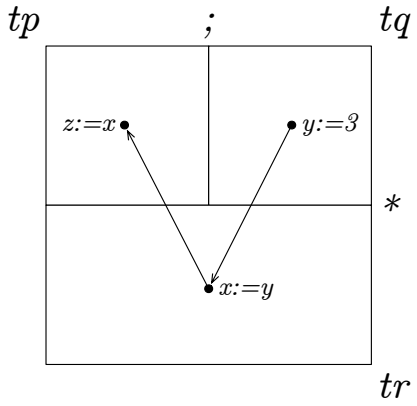

Figure 1: Backward dependency flow

Consider the trace $(tp\,;tq)*tr$ in Fig. 1. In our definition, an event in tp can depend on an event of tq through a chain of dependencies in the concurrent thread tr. The events in Fig. 1 are labeled with assignment statements to show how such a trace might arise. As a result of standard optimisations, this apparently paradoxical data flow—in which z may take the value 3—occurs in standard computers of the present day due to optimizations that result in relaxed memory consistency guarantees. Thus, our theory faithfully represents the problematic features of the real world. It is therefore surprising and encouraging that the model validates all the familiar proof rules of sequential and concurrent reasoning about programs [5, 7], as we show.

3. Program algebra

To facilitate comparison with (original) separation logic, in this section we use notations of propositional logic to represent the corresponding operations on sets. For example, $P \vee Q$ represents a nondeterministic choice between P and Q, and can easily be executed by executing either one of them. $P \wedge Q$ specifies a conjunction of two desirable properties of a program; it may be false, and therefore cannot in general be executed. Nevertheless, conjunction is an essential connective for modular specification of programs. We define the special predicate *false* as the empty set of traces (which cannot be implemented because no trace satisfies it), and *skip* as the set that contains only the empty

trace (which is easily implemented by doing nothing). For predicate P and trace tp, we write $P(tp)$ to indicate $tp \in P$, and say in this case that tp satisfies P.

The inclusion $P \subseteq Q$ can be interpreted as a refinement ordering:

$$P \models Q \equiv_{\mathbf{def}} P \subseteq Q.$$

It means that every trace observed of P is also a possible trace of Q. Consequently, if P can be implemented by executing a particular trace, then so can Q by executing that same trace; and if P is incorrect, then so is Q. As result, Q can be implemented by running P instead.

It is easy to show that the (;) operator has all the familiar properties: it is associative, distributes across disjunction, and is monotonic. *skip* is its unit and *false* is a zero. Finite iteration of a program P is given by the least fixpoint of the function $\lambda X.(\mathit{skip} \vee (X; P))$, defined in accordance with the Knaster-Tarski theorem. Trace sets are thus a model of Kleene algebra [8], where the additive and multiplicative operations are given respectively by disjunction and sequential composition (or, equally well, disjunction and concurrent composition). Similar laws hold for the $(*)$ operator, and it is commutative as well. Indeed, the definitions of $P * Q$ and $P ; Q$ are instances of general model theories of separation logic and bunched logic [12, 3, 2], and thus inherit all of the general properties implied by these theories. In the sequel we describe properties of the model that concern *interaction* between the connectives.

Theorem 1 (Exchange)**.** $(P * Q) ; (P' * Q') \models (P ; P') * (Q ; Q')$

Intuitively, the exchange law above holds because the antecedent restricts dependencies of P' and Q' on both P and Q, whereas the consequent only restricts dependencies of Q' on Q and of P' on P. The laws below are easily derived from the exchange law by substituting *skip* for some of the operands.

Corollary 1.
1. $P ; Q \models P * Q$
2. $P ; (Q * R) \models (P ; Q) * R$
3. $(P * Q) ; R \models P * (Q ; R)$

4. Hoare Logic

The familiar assertional triple $P \ \{ \ Q \ \} \ R$ over predicates P,Q and R is defined as follows:

$$P \ \{ \ Q \ \} \ R \equiv_{\mathbf{def}} (P ; Q) \models R.$$

P describes what has happened before Q starts and R describes what has happened when Q has finished. The more familiar kind of single-state assertions describe the history abstractly as the set of traces that end in a state satisfying the assertion. The usual axioms of assertional reasoning [5] are easily proved sound.

Theorem 2. 1. $P \ \{ \ Q \ \} \ R \ \& \ P \ \{ \ Q \ \} \ R' \ \Rightarrow \ P \ \{ \ Q \ \} \ R \wedge R'$

2. $P \ \{\ Q\ \}\ R \ \&\ P' \ \{\ Q\ \}\ R \ \Rightarrow\ P \vee P' \ \{\ Q\ \}\ R$
3. $P \ \{\ Q\ \}\ S \ \&\ S \ \{\ Q'\ \}\ R \ \Rightarrow\ P \ \{\ Q\,;Q'\ \}\ R$
4. $P \ \{\ Q\ \}\ R \ \&\ P \ \{\ Q'\ \}\ R \ \Rightarrow\ P \ \{\ Q \vee Q'\ \}\ R$
5. $P \ \{\ Q\ \}\ R \ \Rightarrow\ F * P \ \{\ Q\ \}\ F * R$
6. $P \ \{\ Q\ \}\ R \ \&\ P' \ \{\ Q'\ \}\ R' \ \Rightarrow\ P * P' \ \{\ Q * Q'\ \}\ R * R'$

The last two laws are reminiscent of two of the basic axioms of separation logic [9]. However, the standard interpretation of the separating conjunction is radically different from ours.

We define the weakest precondition of program Q and postcondition R:

$$(Q\,{-};R)(tr) \ \equiv_{\mathbf{def}}\ \forall tq\,.\,Q(tq)\ \&\ (tr\,;tq)\ \text{defined}\ \Rightarrow\ R(tr\,;tq).$$

Informally, a trace satisfies $(Q\,{-};R)$ if, whenever followed by a trace of Q, the combined trace satisfies R. The following theorem states in terms of Hoare triples that $(Q\,{-};R)$ is a pre-condition of R under program Q, and that it is the weakest such condition.

Theorem 3 (Galois adjoint)**.** $P \ \{\ Q\ \}\ R \ \Leftrightarrow\ P \models (Q\,{-};R)$

The $(*)$ operator has a similarly defined adjoint called the magic wand, usually written $(-\!\!*)$.

5. The Rely/Guarantee Calculus

A predicate is called *acyclic* when all of its traces are acyclic. Some theorems in this section require this assumption to ensure linearizeability. Predicate G is called an *invariant* when every event of a trace that satisfies G also itself satisfies G:

$$G \text{ invariant } \equiv_{\mathbf{def}}\ \forall tr\,.\,(G(tr)\ \Leftrightarrow\ \forall e \in tr\,.\,G(\{e\})).$$

Invariants are also satisfied by the empty trace, which informally means that an invariant can be satisfied by doing nothing. The strongest invariant implied by both G and G' is $(G \wedge G')$; the predicate $G \nabla G'$ defined below is the weakest invariant that implies both G and G':

$$G \nabla G'(tr)\ \equiv_{\mathbf{def}}\ \forall e \in tr\,.\,G(\{e\}) \vee G'(\{e\}).$$

It is easy to see that a predicate G is an invariant iff $G = G \nabla G$; other facts are collected below.

Theorem 4. *For invariant G:*

1. $G(tp \cup tq)$ *iff* $G(tp)$ *and* $G(tq)$
2. $G\,;G = G * G = G$
3. $G * [\ell] = G\,;[\ell]\,;G$*, when* $G * [\ell]$ *is acyclic.*

As in the Jones rely/guarantee calculus, invariants are used to describe ways in which one process is permitted to interfere with another. A rely condition is an invariant that describes the assumptions a process makes about interference from its environment during execution; similarly, a guarantee condition is an invariant that describes the guarantee that a process makes regarding its own interference with the environment. Satisfaction of the invariant condition G by process Q is simply expressed by the implication $Q \models G$. Such a judgement can be proved by the following calculus.

Theorem 5. *For invariant G:*

1. $Q \models G \;\&\; Q' \models G' \;\Rightarrow\; (Q * Q') \models (G \nabla G')$
2. $Q \models G \;\&\; Q' \models G' \;\Rightarrow\; (Q\,;Q') \models (G \nabla G')$.

The *Jones quintuple* $P\ R\ \{\ Q\ \}\ G\ S$ is a partial correctness specification of program Q in the presence of interference from other threads. It allows the program Q to rely on the environment to satisfy the invariant R, and in turn guarantees condition G. Q also satisfies postcondition S on assumption of precondition P:

$$P\ R\ \{\ Q\ \}\ G\ S \equiv_{\mathbf{def}} P\ \{\ R * Q\ \}\ S\ \&\ Q \models G.$$

The base rule of the Jones calculus reduces concurrent reasoning to sequential reasoning; reasoning about concurrent composition is reduced to reasoning about individual processes.

Theorem 6. $P\ R\ \{\ [\ell]\ \}\ G\ S \Leftrightarrow P\ \{\ R\,;[\ell]\,;R\ \}\ S\ \&\ [\ell] \models G$, *when* $R * [\ell]$ *is acyclic.*

Theorem 7 (Concurrency)**.** $P\ R\ \{\ Q\ \}\ G\ S\ \&\ P'\ R'\ \{\ Q'\ \}\ G'\ S' \Rightarrow$ $(P \wedge P')\ (R \wedge R')\ \{\ Q * Q'\ \}\ (G \nabla G')\ (S \wedge S')$, *when* $G' \models R$ *and* $G \models R'$.

Proof. $Q * Q' \models G \nabla G'$ follows from Thm. 5. Below we show $(P \wedge P')\ \{\ (R \wedge R') * (Q * Q')\ \}\ S$. Similarly, we can show $(P \wedge P')\ \{\ (R \wedge R') * (Q * Q')\ \}\ S'$, from which the conclusion follows from Thm. 2.

$$\begin{array}{ll} & (P \wedge P')\,;((R \wedge R') * (Q * Q')) \\ \models & \{\ (;) \text{ and } (*) \text{ are monotonic } \} \\ & P\,;(R * (Q * Q')) \\ \models & \{\ Q' \models G' \text{ and } G' \models R\ \} \\ & P\,;(R * (Q * R)) \\ \models & \{\ \text{associativity and commutativity of } (*)\ \} \\ & P\,;(R * R * Q) \\ \models & \{\ R \text{ is invariant, so } R * R = R \text{ by Thm. 4}\ \} \\ & P\,;(R * Q) \\ \models & \{\ P\,;(R * Q) \models S \text{ by assumption}\ \} \\ & S. \end{array}$$

□

The Jones rule for sequential composition is:

$$P\ R\ \{\ Q\ \}\ G\ S\ \&\ S\ R'\ \{\ Q'\ \}\ G'\ S'\ \Rightarrow\ P\ (R \wedge R')\ \{\ Q\,;Q'\ \}\ (G \nabla G')\ S'.$$

To prove this, we must show that $Q\,;Q' \models G \nabla G'$, which follows from Thm. 5, and also that $P\ \{\ (R \wedge R') * (Q\,;Q')\ \}\ S$. But this assertion is invalid. Consider the following model: $P = \{\emptyset\}$, $Q = \{\{q\}\}$, $Q' = \{\{q'\}\}$, $R = R' = \{\{r\}, \emptyset\}$, $S = \{\{r, q\}, \{q\}\}$, and $S' = \{\{q, q'\}\}$, with $q' \to r \to q$. One can check that the antecedents hold in this model, and also that $\{r, q, q'\}$ satisfies $P\,;((R \wedge R') * (Q\,;Q'))$, but not S'. The countermodel is like the example in Fig. 1 in that there is no direct dependency between q and q', but interference from the environment yields an indirect dependency from the second operand to the first.

The Jones rule is of course valid for normal strong sequential composition, since all the events of the second operand would be forced by control dependency to be executed after those of the first operand. The anomalous dependency is then ruled out by acyclicity. But this would sacrifice all opportunity for standard optimisations. All that is necessary is to ensure that the "critical" events in the trace of $P\,;Q$ are connected by a control dependency. In practice , critical events are protected by a semaphore; and the definition of the acquisition and release of semaphores requires that they be linearly ordered by a control arrow.

To prove the weak sequential composition rule, we formalize our assumptions as follows. We write $\leftarrow$ and $\xrightarrow{+}$ for the inverse and transitive closure of the $\to$ relation, respectively; $\leftrightarrow$ and $\xleftrightarrow{+}$ for $\to \cup \leftarrow$ and $\xrightarrow{+} \cup \xleftarrow{+}$. To restrict a relation to trace tp, we write, e.g., $\xleftrightarrow{+}_{tp}$. We say an event $p \in tp$ is *critical to* tr when, for some event $r \in tr$, $p \leftrightarrow r$. In established parlance, traces with critical events are called critical regions. We say tp is *protected from* tr if every pair of events $p, p' \in tp$ that are critical to tr are connected: $p \xleftrightarrow{+}_{tp} p'$. Finally, P is protected from R when each $tp \in P$ is protected from each $tr \in R$, and $P * R$ is acyclic.

Theorem 8. $P\ R\ \{\ Q\ \}\ G\ S\ \&\ S\ R'\ \{\ Q'\ \}\ G'\ S'\ \Rightarrow$
$P\ (R \wedge R')\ \{\ Q\,;Q'\ \}\ (G \nabla G')\ S'$, *when* $(Q\,;Q')$ *is protected from* $(R \wedge R')$.

In Fig. 1, the two events in the trace $(tp\,;tq)$ are both critical to tr. To protect them, it is necessary to connect these critical events by a chain of dependencies. Then the possibility of a backward dependency is ruled out by acyclicity.

Acknowledgements. The authors thank Josh Berdine, Philippa Gardner, Aleksander Nanevski, Viktor Vafeiadis, Glyn Winskel, Hongseok Yang and the members of IFIP WG 2.3 for their comments.

References

[1] L. Caires and L. Cardelli. A spatial logic for concurrency (part i). *Inf. Comput.*, 186(2):194–235, 2003.

[2] C. Calcagno, P. Gardner, and U. Zarfaty. Context logic as modal logic: completeness and parametric inexpressivity. In M. Hofmann and M. Felleisen, editors, *POPL*, pages 123–134. ACM, 2007.

[3] C. Calcagno, P. W. O'Hearn, and H. Yang. Local action and abstract separation logic. In *LICS*, pages 366–378. IEEE Computer Society, 2007.

[4] L. Cardelli, P. Gardner, and G. Ghelli. A spatial logic for querying graphs. In P. Widmayer, F. T. Ruiz, R. M. Bueno, M. Hennessy, S. Eidenbenz, and R. Conejo, editors, *ICALP*, volume 2380 of *LNCS*, pages 597–610. Springer, 2002.

[5] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[6] C. A. R. Hoare, I. Wehrman, and P. W. O'Hearn. Graphical models of separation logic. In *Engineering Methods and Tools for Software Safety and Security*. Summer School Marktoberdorf, IOS Press, 2009. Online at `http://www.cs.utexas.edu/~iwehrman/pub/marktoberdorf-2008.pdf`.

[7] C. B. Jones. *Development methods for computer programs including a notion of interference*. PhD thesis, Oxford University, 1981.

[8] D. Kozen. On Kleene algebras and closed semirings. In *Proceedings, Math. Found. of Comput. Sci.*, volume 452 of *LNCS*, pages 26–47. Springer-Verlag, 1990.

[9] P. W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.

[10] P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In L. Fribourg, editor, *CSL*, volume 2142 of *LNCS*, pages 1–19. Springer, 2001.

[11] V. R. Pratt. The pomset model of parallel processes: Unifying the temporal and the spatial. In S. D. Brookes, A. W. Roscoe, and G. Winskel, editors, *Seminar on Concurrency*, volume 197 of *LNCS*, pages 180–196. Springer, 1984.

[12] D. J. Pym, P. W. O'Hearn, and H. Yang. Possible worlds and resources: the semantics of BI. *Theor. Comput. Sci.*, 315(1):257–305, 2004.

[13] D. J. Pym and C. M. N. Tofts. A calculus and logic of resources and processes. *Formal Asp. Comput.*, 18(4):495–517, 2006.

[14] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.

[15] A. W. Roscoe. *The theory and practice of concurrency*. Prentice Hall, 1998.

[16] G. Winskel. Event structures. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Advances in Petri Nets*, volume 255 of *LNCS*, pages 325–392. Springer, 1986.