Unifying models of data flow

Tony HOARE [a] and John WICKERSON [b]
[a] *Microsoft Research Cambridge*
[b] *University of Cambridge Computer Laboratory*

Abstract. We propose a model of computation, based on data flow, that unifies several disparate programming phenomena, including local and shared variables, synchronised and buffered communication, reliable and unreliable channels, dynamic and static allocation, explicit and garbage-collected disposal, fine-grained and coarse-grained concurrency, and weakly and strongly consistent memory.

Keywords. Unifying theories, concurrency, communication, weak memory

1. Introduction

A unifying model is one that generalises a diverse range of more specific theories, each of them applicable to a range of phenomena in the real world. The original models turn out to be special cases of the unifying model. The individual merits of each specific model are clarified and substantiated by the unification, and they continue to be useful, perhaps in combination with other theories, in their special areas of application.

The aim of this article is to propose a unifying model for a collection of important computing concepts that are widely used in computer programming. Currently accepted theories for these concepts are often expressed by an operational semantics, describing abstractly how they can be implemented in the in the context of a particular programming language. Such implementations are an excellent method of differentiating concepts, features, and of complete programming languages.

Since our goal is the opposite of that, we take a different approach. The similarities of specific theories are codified by the properties that they all share, and their differences are demonstrated by counterexamples to these properties. Each property aims to express clearly the primary purpose of the concept and its behaviour, and ignores completely the ways in which it might be implemented, either in software or by the hardware of a physical device. Each property says as little as possible about the concept that it describes, and thereby admits a whole range of variations, which can be explored by adding further properties individually. Every theorem proved from the earlier properties remains valid for all of the subsequent variations.

Our style of presentation is therefore similar to that of a well-structured mathematical textbook. We hope that it will not be too unfamiliar and uncomfortable to readers who are accustomed to seeing a complete characterisation of each concept at the place where it is first defined.

We take data flow as our primitive concept. Data held in a central computer memory flows across the interval of time separating an assignment of a value to a memory location from an access of the value assigned. Data communicated on a channel flows across the interval of space separating two components of a real or simulated computer network. Acknowledgement signals are treated as communication of a null message, and synchronisation can be modelled as acknowledgement in both directions. This insight enables us treat all these forms of flow uniformly.

For both memory and communication, we will classify several variations of each kind of resource. In the case of computer memory, we will deal with variables that are private to an individual thread and those which are shared between multiple threads, which run concurrently by interleaving accesses to shared memory at varying levels of granularity. The hardware of the memory may conform to strong or weak rules of consistency.

Variations of our model of communication include channels that are buffered or synchronised, stuttering or lossy, and overtaking or order-preserving. There are various methods of allocation and disposal of computer resources, including nested allocation of data declared local to a block, as well as data dynamically allocated on a heap. Heap data is either explicitly disposed by the program or recovered automatically in an implementation, for example by a garbage collector. All the models listed above will cover both the sequential and concurrent use of resources, at varying granularities of atomicity.

Related Work

Wehrman et al. [4] complements the current article by providing a unifying model of control flow. Like our data flow model, the control flow model is quite weak, yet it is strong enough to justify the standard methods of verifying both sequential and concurrent programs, in particular Hoare logic [1] and Jones' Rely/Guarantee reasoning [2]. It is hoped that the unifying model will also justify other familiar forms of semantic definition, including operational, algebraic and denotational; this remains as a prospect for future research.

2. Traces

The main ideas will be conveyed pictorially by a graph recording a trace of all events that have occurred in a particular execution of a particular program. A program can be thought of as a set of these traces, each trace representing one possible execution. Each event in a trace will be pictured as a box, often containing an indication of the nature of the event. A trace of a complete program can be split into separate traces, one for each computer resource used by the program. Such a trace contains just the events in which that resource has engaged.

The trace also records the data that flows between its events. The occurrence of flow will be drawn as an arrow between the boxes that are the source and target of the flow. This establishes a dependency relation between the target and the source, in that occurrence of the source event is necessary for the target event to occur. Certainly the target event cannot possibly occur before the source event.

Figure 1 shows a first example of a trace of a single resource that is used sequentially. It contains exactly five events drawn as boxes. Each box contains a label indicating the nature of the event that it represents. For example, νs labels the event that allocates the

$$\begin{tikzcd}
\boxed{\nu s} \arrow[r, tail] & \boxed{s} \arrow[r, tail] & \boxed{s} \arrow[r, tail] & \boxed{s} \arrow[r, tail] & \boxed{\delta s}
\end{tikzcd}$$

Figure 1. A sequential trace

resource s, and δs labels the event that disposes it. All the events are labelled by the name of the resource itself.

In the figure, the events are connected in a series by four arrows, indicating the dependency or data flow between them. Each event except the first depends upon its unique predecessor, and each event except the last has a unique successor that depends uon it. Because of the linear chain of dependencies, each event can only occur after the previous event and before the next event. There is no possibility of concurrent execution of events.

Obviously, the graph in Fig. 1 is just one example of a general sequential design pattern, which happens to have length five. We need some way of defining the general pattern of a sequential resource, to allow instances of any length.

To define precisely the set of traces that conform to our pictures, we will supplement them with formal statements expressed generally in the notations of the relation calculus [3], as described in the next section. For example, the relational formula $\nu\,(s \rightarrow s)^*\,\delta$ uses relational composition and the Kleene iteration operator to indicate a chain of arbitrary length.

Here is a more formal definition of our concept of a data flow graph. It is a mathematical triple, comprising:

$$\begin{array}{rcll}
\square & & & \text{(the carrier set of events)} \\
\rightarrow & \subseteq & \square \times \square & \text{(a relation between events)} \\
s, c, x, \delta, \ldots & \subseteq & \square & \text{(a collection of subsets of events)}
\end{array}$$

The first component is the set of events, drawn as boxes. The second component is a relation between events, formally defined as a set of pairs of boxes. It is drawn as an arrow between each pair of related boxes. The third component is a collection of subsets of the events of the graph. They are drawn by labelling each box with the names of the subsets to which it belongs.

3. Relation calculus

Here are some useful standard concepts and properties of the relation calculus.

- If two elements are related by a relation m, they are usually written on either side of it:

$$e\;m\;f \;\stackrel{\text{def}}{=}\; (e, f) \in m.$$

- The composition of two relations m and n will usually be denoted by simple juxtaposition; that is,

$$e\;(m\,n)\;f \;\stackrel{\text{def}}{=}\; \exists g.\, e\;m\;g \wedge g\;n\;f.$$

- The identity relation, which holds between anything and itself, is denoted by *Id*:

$$e\ \mathit{Id}\ f \ \stackrel{\text{def}}{=}\ e = f.$$

- The universal relation, which holds between every pair of elements is denoted by $\mathbb{U}$:

$$e\ \mathbb{U}\ f \ \stackrel{\text{def}}{=}\ \textsf{true}.$$

- The converse of a relation, which holds between pairs written in the opposite order, is generally denoted by superscript cup, or sometimes by reversing the symbol denoting the relation:

$$\begin{aligned} e \leftarrow f &\ \stackrel{\text{def}}{=}\ f \rightarrow e \\ e\ (m^{\cup})\ f &\ \stackrel{\text{def}}{=}\ f\ m\ e. \end{aligned}$$

- The Kleene star will denote the iteration of a relation; it stands for the composition of any number of instances of the relation. A positive iteration is similarly defined. A relation defined by iteration is always transitive and reflexive, so we denote it by a traditional ordering symbol $\leq\ \stackrel{\text{def}}{=}\ (\rightarrow)^*$:

$$\begin{aligned} (m)^* &\ \stackrel{\text{def}}{=}\ \mathit{Id} \cup m \cup mm \cup mmm \cup \ldots \\ (m)^+ &\ \stackrel{\text{def}}{=}\ m \cup mm \cup mmm \cup \ldots. \end{aligned}$$

Many important properties of relations can be defined using the operations defined above. In our interpretation of the dependency relation, a cycle of dependency would require all events in the cycle to occur simultaneously. In some graphs, we may wish to exclude this possibility, and require that an event is synchronised only with itself. More formally, in such a graph, the ordering relation $\leq$ will be a partial order, in that it satisfies the antisymmetry law:

Definition 3.1 (Antisymmetry). $\leq$ is antisymmetric iff:

$$(\leq \cap \geq) \subseteq \mathit{Id}$$

or, expanded to predicate calculus,

$$\forall e, f.\, e \leq f \wedge e \geq f \ \Rightarrow\ e = f.$$

Relational calculus can distinguish a relation that is a (partial) function. If the converse of a function is composed with the function itself, the result is always a subset of the identity relation:

Definition 3.2 (Partial function). m is a partial function iff:

$$(m^{\cup}\, m) \subseteq \mathit{Id}$$

or, expanded to predicate calculus,

$$\forall e, f.\, e\ m\ f \wedge e\ m\ g \ \Rightarrow\ f = g.$$

In both these examples, the meaning can be fully explained in predicate calculus by introducing variables and quantifiers. In both cases, the brevity of the relational notation makes algebraic reasoning much simpler than it is in the more explicit predicate notation. That is useful once one gets used to it.

3.1. Using relation calculus to describe traces

A resource is represented by the set of events in which it has engaged. We will use a lower case letter to stand for a set of events that have occurred in a trace of a particular resource. We shall write c, d, etc. for channels, x, y, etc. for variables, and r, s, etc. for general resources.

Following standard practice, we represent a set as a special kind of relation. It is the restriction of the identity relation to elements of that set. That is:

$$e\;s\;f \quad \overset{\text{def}}{=} \quad e \in s \wedge e = f$$

where s is a relation on the left-hand side, but a set on the right-hand side. The advantage of this convention is that we can represent the intersection of two sets by relational composition:

$$s \cap t = s\,t.$$

An arrow composed on the left with a set restricts the domain of the arrow to that set, and composition with a set on the right restricts the codomain similarly. As a result, a chain of alternating arrows and sets of events will represent a path in a graph that passes through the relevant sequence of event sets. For instance, $s \rightarrow t$ is a relation containing all arrows with source in s and target in t. We will continue to use set symbols as sets, where this is more convenient. It is possible to get used to the ambiguity.

As a first example of the use of relational abbreviations, we return to the concept of a sequentially reusable resource s. All the events in which the resource engages are linked by a single chain of dependency. One property of such a chain is that there are no branching points. So consider a path that starts in the set s, moves rightward to another member of s, and then takes a step backwards. Such a path is described by the composition $s \rightarrow s \leftarrow s$. The law we want is that such a path always leads back to the event at which it started. The case is similar for a path that moves leftward and then rightward. Each event in s has at most one successor and at most one predecessor in the direct dependency relation:

$$(s \leftarrow s \rightarrow s) \subseteq \mathit{Id} \qquad \text{and} \qquad (s \rightarrow s \leftarrow s) \subseteq \mathit{Id}.$$

In addition to this functional property, we need to state that any two events in s are connected by indirect dependency in one direction or the other. In other words, the dependency ordering (restricted to the set s) is a total ordering on s. This rules out gaps in the chain of dependency:

$$(s \leftarrow s)^* \cup (s \rightarrow s)^* \quad = \quad s \;\mathbb{U}\; s.$$

We are already using italic letters to denote sets of events involving the same resource. We also need to distinguish different types of event, for which we introduce symbols such as ν, which denotes the set of all allocation events, and δ, which is the set of all deletions. For variables, we then distinguish assignments, written $:=$, from fetches (written $=:$) of a previously assigned value. For channels, we distinguish inputs (written $?$) from outputs (written $!$). For semaphores, we distinguish acquisition (written $\Downarrow$) from release (written $\Uparrow$).

Most events will belong to two or more of these sets. For example, the allocation of a resource s belongs both to the set s and to the set ν. We will exploit the identity of composition and intersection of sets to build up notations very suggestive of a programming language; for instance, $x :=$ denotes the set of all assignments to x, while $x =: 5$ is the set of fetches of the value 5 from x. In all cases, the meaning is a set, which of course, in a given trace may be empty, or perhaps contain only one member.

In a diagram, an event box is often labelled by a set to which it belongs. This may identify both the resource involved and the nature of the event. But we will not be obliged to write a complete description of every property of an event. In general, we will concentrate only on what is important.

4. Allocation and disposal

Now we can embark on the main task of constructing our model of data flow. And we start at the beginning, with the important and very general concept of allocation of a resource. The most important property of resource allocation is that it should be the first event in the life of the allocated resource. This property seems to be close to describing the actual purpose of allocation. So we take this as the defining property of allocation. Without loss of generality, we can assume that a resource has a unique allocation event, even if it occurred before the program started, say when the hardware was made.

Definition 4.1 (Properties of allocation)**.** Allocation is the first event of a resource s:

$$s \;\subseteq\; (s \geq \nu s \leq s)$$

and each resource s has exactly one allocation event:

$$|\nu s| \;=\; 1.$$

Similar definitions apply for disposal, where the disposal may occur after termination of the program, say when the hardware is thrown away.

And that is all we want to say about allocation and deletion! We certainly do not want to say anything about methods of implementation, of which there are many. For instance, a global variable of the program is often allocated by the compiler, which gives it a fixed location. A variable declared locally in a block of the program is allocated on a stack. A variable declared as (an attribute of) an object is allocated dynamically on the heap. All of their allocations satisfy the same defining property given in Defn. 4.1.

There are even more ways that disposal may be implemented. For example, a declared variable may be disposed from the stack on exit from the block in which it is declared. A normal compiler scope check ensures that the variable engages in no event af-

$$\begin{tikzcd}\boxed{\nu s} \arrow[r] & \boxed{s \Uparrow} \arrow[r] & \boxed{s \Downarrow} \arrow[r] & \boxed{s \Uparrow} \arrow[r] & \boxed{s \Downarrow} \arrow[r] & \boxed{\delta s}\end{tikzcd}$$

Figure 2. A semaphore

ter block exit. A dynamically allocated variable may be disposed by garbage collection, which detects by scanning and marking that no further use of it will ever be made. It may be disposed by an explicit command in the program, or by the operating system upon completion of the entire program. In the last resort, it may disappear when the computer is switched off or even thrown away.

5. Semaphores

Let us develop our first example (a sequential resource) as a simple Boolean semaphore. The semaphore s shown in Fig. 2 can engage in only two substantive types of operation: an acquisition, denoted by $\Downarrow$, and a release, denoted by $\Uparrow$. After allocation, the first action of this semaphore is a release, which makes the resource available for acquisition by other threads. Any subsequent pair of consecutive actions is then either a release followed by an acquire, or vice-versa. The last action before deletion must be an acquisition. There is a final possibility that there are no actions except allocation and deletion. All of these alternatives are listed as a union on the right hand side of the following inclusion:

$$s \to s \subseteq (\nu \to \Uparrow) \cup (\Uparrow \to \Downarrow) \cup (\Downarrow \to \Uparrow) \cup (\Downarrow \to \delta) \cup (\nu \to \delta)$$

Needless to say, we have described the purpose of the semaphore, without saying anything about its implementation.

6. Fan-in and fan-out

We now move on to the more interesting types of resource that are non-sequential, and permit concurrent execution of some of the events. A simple example is provided in a pure functional (or single assignment) programming language, where the only variables are function parameters. Each call to the function allocates a different new variable, and simultaneously assigns to it the value w of the parameter. This is the only assignment ever made to x. The data flow arrows carry this assigned value to every event in the function body that fetches it. There is no dependency ordering between any of the fetch events; they may be executed in any order, or even concurrently. That is indicated by the absence of arrows between the events.

The behaviour of a parameter is described pictorially in Fig. 3 as a graph. The graph has the shape of a fan-out in hardware. A more formal definition is as follows:

Definition 6.1 (Parameter)**.** Every arrow has the same source:

$$x \to x \leftarrow x \quad \subseteq \quad \mathit{Id}$$

and every arrow must connect the initialising assignment (of which there can be only one, in accordance with Defn. 4.1) to a fetch from the same variable x of the same value w:

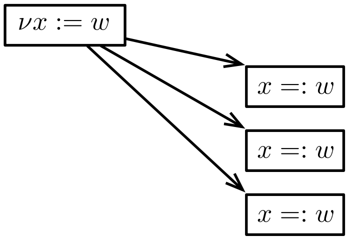

Figure 3. A parameter

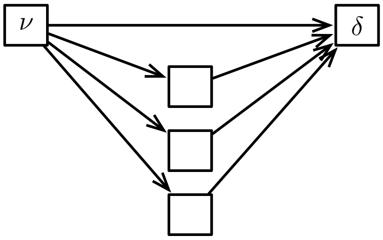

Figure 4. A concurrent resource

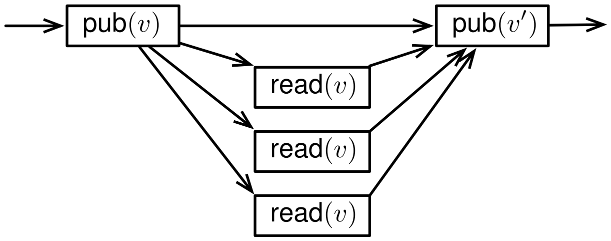

Figure 5. Publication

$$x \to x \;\subseteq\; \bigcup_w (\nu x := w) \to (x =: w).$$

The mirror image of a fan-out is a fan-in. All of its arrows have the same target. Occurrence of the target event depends on occurrence of all the source events: the target event cannot occur any earlier than the latest event on which it depends. The arrows of a fan-in usually transmit acknowledgement signals rather than data values. These signals prevent the event at the target of the arrows from occurring before it is wanted.

Figure 4 presents an example of the use of a fan-out followed by a fan-in: it describes a resource that has been shared among any number of concurrent threads. Each use of the resource is independent of all the others, and they may occur concurrently. Yet there is no prohibition of sequential use, and each thread can use the same resource many times. The boxes have been left empty, and can be filled in many different ways. Allocation and disposal are subject to the usual constraints. Every event in which the resource engages must be preceded by an allocation event, which is the source of the fan-out, and it must be followed by a single disposal event, which is the target of the fan-in.

A familiar example of the use of the concurrent resource pattern is the distributed publication of values over an ‘ether’ to a known set of subscribers. In the simplified diagram shown in Fig. 5, there are only three subscribers. The events at the top of the diagram are the publication of two successive messages. The other boxes stand for the reads by the subscribers of the most recent value published. Delivery of the value to all subscribers is guaranteed by an acknowledgement signal, sent by each subscriber to the publisher after reading. In this simple example, the publisher must not publish the next message until all the acknowledgement signals have been received. That is ensured by the fan-in arrows shown in the diagram.

We now come to a much more familiar example, which uses the same pattern as the publish-subscribe protocol. Figure 6 shows part of the behaviour of an ordinary program

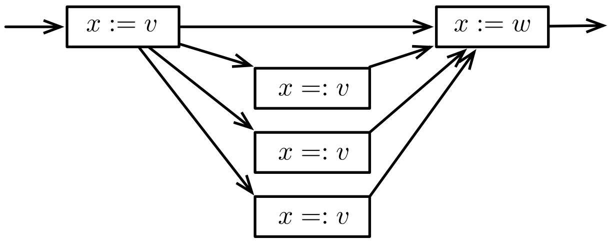

Figure 6. Assignment

variable allocated in the central memory of a computer. The assignments to the variable follow the pattern of a sequential resource, linked by a chain of dependency arrows. Just two of them are shown at the top of the diagram.

The fetches are connected to the preceding assignment by a data dependency along which the assigned and fetched value can flow. The fetches are also connected to the following assignment by a fan-in. In contrast to the implementation of the publish/subscribe pattern, the implementation of a variable in memory requires no acknowledgement signals at run-time. This is because the hardware of a strongly-consistent memory ensures firstly that the next assignment to the variable will immediately overwrite the previously stored value, and secondly that each read will read the result only of the most recent assignment. Thus it is logically impossible for the next assignment to occur until all reads of the previous assignment have taken place. This is the reason for the dependencies that are recorded by the arrows of the fan-in.

We have seen the same fan-out/fan-in pattern of arrows re-used three times to describe three different computing phenomena: the sharing of resources among concurrent threads, the publication and acknowledgement of data published by broadcast, and the assignment of values to variables in memory. This is a good example of the kind of unification that we seek in our theories. It is achieved by concentrating on the purpose and logic of the behaviour of a computer executing a program, and by ignoring all issues of implementation.

7. The token game

Our traces can be interpreted as Petri nets [5]. Their operational significance can be illustrated in the same way as Petri nets, by a token game.

Figure 7(a) shows a token residing on an arrow. It passes along the arrow, and through the box at the target of the arrow, following rules that are illustrated in the rest of the figure. As it passes through a box, the token splits into enough parts to pass along all the arrows leading out of the box. When all the arrows leading into a box are filled with tokens, then all the tokens can pass simultaneously through the box, and appear as tokens on all the outgoing arrows of the box. This is called the firing of the box. In Fig. 7(b), all the fetch boxes are ready to fire. At any time, any subset of the boxes that are ready to fire may fire simultaneously. In Fig. 7(c), two of the fetches have fired, but not the third. As a result, the final assignment of the slide is not ready to fire. So the only possible next event on this diagram must be the occurrence of the third fetch. In Fig. 7(d), there are now enough tokens to pass together through the final assignment. In doing so,

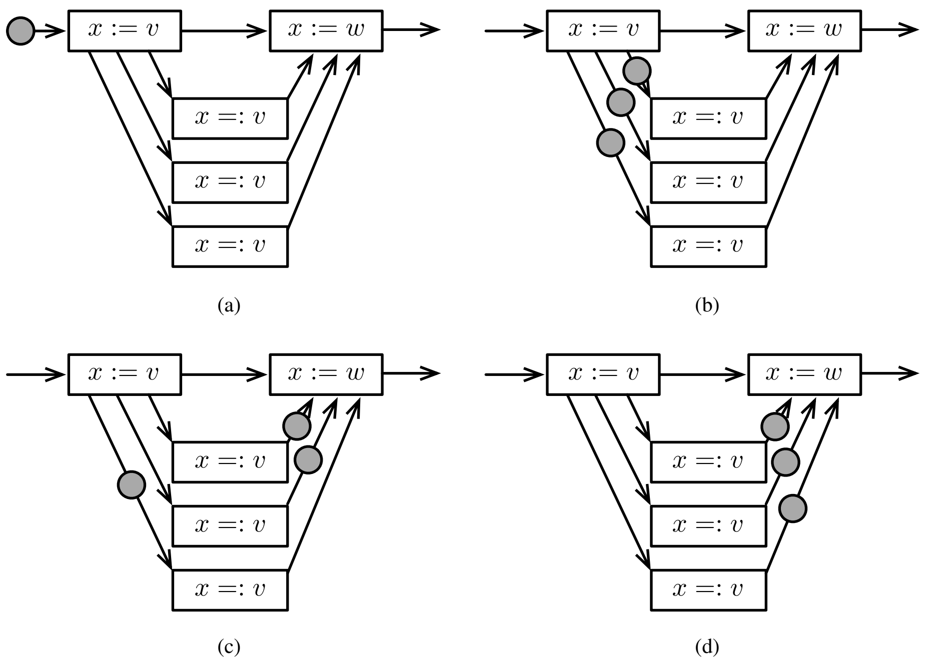

Figure 7. The token game

they merge, and re-divide again if necessary, to pass along all the outgoing arrows of the event.

Note the sharp distinction between our diagrams and those that illustrate a finite state automaton. In the latter, there is only a single token, which never splits. A box with many outgoing arrows therefore stands for a choice between them. The whole token then travels along the chosen arrow. Thus all graphs described by a finite state automaton are sequential.

Our diagrams are intended to record a single trace of a single execution, in which all choices have already been made. That is why we can use multiple outgoing arrows to indicate multiple concurrent successors to an event. We model choice in a program as giving rise to a whole set of traces.

8. Variables

The more elaborate diagram in Fig. 8 gives an example of an entire (short) history of events involving a particular variable, declared locally or allocated dynamically by the program. The assignments are strictly ordered, like a sequential resource. Between any consecutive pair of assignments, the fetches of the value may take place in any order, or even concurrently.

As before, the inclusion below constrains the source and target of all the direct dependency arrows within the resource x:

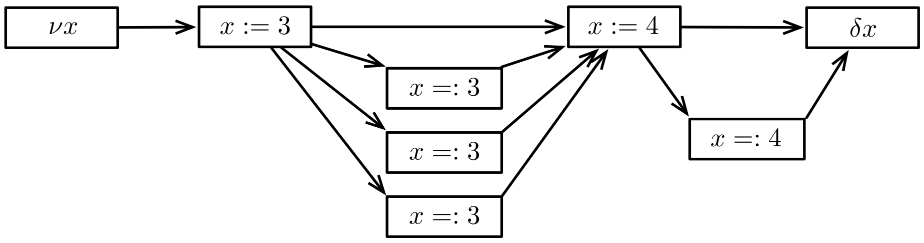

Figure 8. A variable

$$x \rightarrow x \subseteq (\nu \rightarrow :=) \cup (\nu \rightarrow =:) \cup (:= \rightarrow :=) \cup (:= \rightarrow =:) \cup \\ (=: \rightarrow :=) \cup (\nu \rightarrow \delta) \cup (:= \rightarrow \delta) \cup (=: \rightarrow \delta)$$

In our behavioural diagrams, the closed triangles add a great deal of implicit information about the way in which two arrows or paths in the trace of a variable share a common source or a common destination. Equation 1 below makes this information formally explicit: a path which goes all round the triangle must lead back to the original starting position. For instance, consider Fig. 8 and suppose we start our path at the "$x := 3$" event. We follow an arrow down to one of the fetches and then follow an acknowledgement arrow back up to the next assignment (which is $x := 4$ in this case, but could in general be a disposal). Finally, we follow the top arrow backward to the preceding assignment. The closure property of the triangle states that this anti-clockwise circular path ends up in the same place that it began.

$$(:=) \rightarrow (=:) \rightarrow (:= \cup\, \delta) \leftarrow (:=) \quad \subseteq \quad (:=) \tag{1}$$

Exactly the same information is conveyed in the algebraic equation in the equation above. The left-hand side of the inclusion describes the set of paths that start with any assignment, and then follow an arrow to a fetch, then follow an arrow to another assignment (or disposal), and finally follow an arrow backwards to an assignment again. It describes exactly all anticlockwise paths, starting with an assignment, that go around any triangle of the shape shown. On the right side of the inclusion there is a null path that starts and ends immediately with the identity relation over assignments. The inclusion itself therefore says that every anticlockwise path described on the left will lead back to the same assignment that it started with. The inclusion is of course also valid for an assignment that is never fetched, because the left hand side is then empty.

The concept of a closed triangle is very similar to that which we have already encountered in the definition of a sequential resource. There, we saw that a path following an arrow forward from a node, and then following an arrow backward, must end up at its starting place. Here, the length of the path is just one segment longer.

There another equivalent way of expressing the same closed property. It is obtained by traversing the same path in a clockwise order, rather than anticlockwise. Obviously, the arrows are traversed in the opposite order, and in the opposite direction.

$$(:=) \rightarrow (:= \cup\, \delta) \leftarrow (=:) \leftarrow (:=) \quad \subseteq \quad (:=)$$

Actually this clockwise formulation is equivalent to the anticlockwise path of Eq. 1.

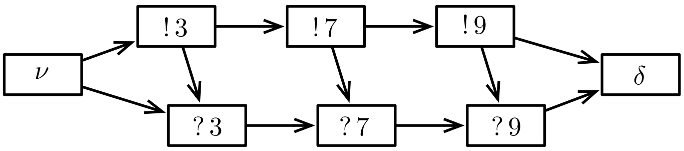

Figure 9. A channel

Another equivalent way to express this property involves starting the path at the second assignment event, and again proceeding either clockwise or anticlockwise. But the property does not hold for paths that start with the fetch event. This is because the fan-in or the fan-out that connects the fetch allows a choice at the final step in the path, and the choice may fall on an event different from the starting point.

9. Communication channels

The other main method of implementing data flow is by communication along channels that connect components of a distributed system. (Our theory will apply equally to simulated communication between separate processes executing as threads on the same computer.) As mentioned earlier, we will distinguish an outputting event by labelling it with an exclamation mark, and an inputting event by a query. Obviously, every successful input event depends on the successful occurrence of the particular output that defines the message that it reads.

Just like assignments to a variable, the output operations on the same channel are sequentially ordered by control dependency, as are all the input operations. However, in a fully buffered channel, there is no dependency of the outputs on the inputs. The implementation of such a channel is expected to interpose a buffer of arbitrary depth in the channel. As a result, the sequence of outputs that have happened so far may get arbitrarily far ahead of the inputs.

Figure 9 shows a pattern for the entire (short) history of all events associated with a single channel. The first event is the allocation of a new channel. This simultaneously allocates both the output end of the channel and the input end. Similarly, the event that disposes the channel comes after both the last output and the last input. The history shows successful communication of three values.

This example illustrates a behaviour that would probably be regarded as desirable for any channel. However, various kinds of unreliable channel may exhibit some less desirable behaviour instead, as we shall shortly describe.

Figure 9 provides two more examples of closed triangles. At the beginning, the first output after allocation provides the message read by the first input. Similarly, the message read by the last input is provided by the last output. In this case, all the arrows are functions in both directions, and the closed triangle property holds independently of both the starting point and the direction of the path. In addition to the closed triangles, we see two examples of closed rectangles. Because there is no fan-out or fan-in, the closure property holds for all starting points.

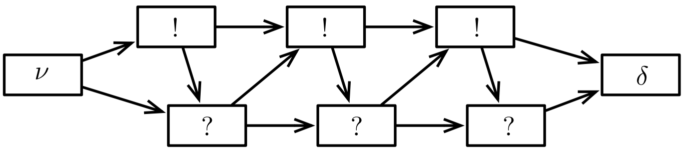

Figure 10. A singly-buffered channel

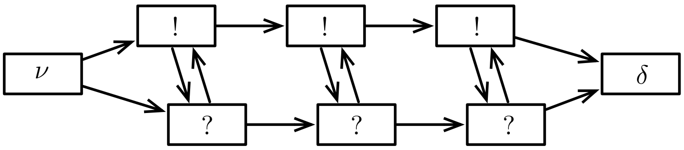

Figure 11. A zero-buffered/synchronised channel

9.1. Buffered channels

Figure 9 places no limit on how many outputs can occur before the first of their corresponding inputs. As mentioned before, this requires implementation with the aid of an unbounded buffer, storing each output value until its corresponding input is performed. Some channels place a limit on the size of the buffer used to implement this asynchrony. The limit introduces a control dependency from an input to the output which first needs to use the buffer space freed by the input.

Figure 10 shows a channel which is limited to single buffering. The additional arrows ensure that an output cannot occur until the previously output value has been actually input at the other end of the channel. One can see from this diagram a certain similarity between a stored variable and a singly-buffered channel.

A synchronised channel is one that has no buffer at all. Each input is synchronised with the corresponding output. Figure 11 shows the synchronisation signals, indicating that the input and the output depend on each other in a cycle. Since no event can occur before any event that it depends on, the only way of executing a dependency cycle of events is to execute all of them simultaneously – which is exactly we intend to say in this case. Note that every area in this diagram that is enclosed by a cyclic path is closed in the same way as the triangles and rectangles of previous channel diagrams.

Most programming languages put severe restrictions on the size and nature of the dependency cycles that are allowed. Any violation of the restrictions is punished by deadlock. Let us not pursue this important issue any further here.

9.2. Badly-behaved channels

Up to this point we have modelled the behaviour of entirely reliable channels. We will now use our graphical conventions to define various ways in which a real channel may fall short of this ideal.

A lossy channel is one for which some of the outputs are never input, as shown in Fig. 12. To specify that a channel is non-lossy, it suffices to require that the data flow relation between an output and the corresponding input should be a total relation on outputs. The concept of totality can be defined by three equivalent formal definitions, two in the relation calculus and one in the predicate calculus:

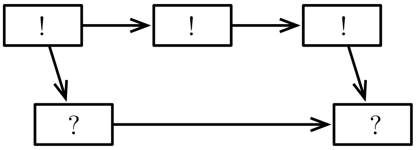

Figure 12. A lossy channel

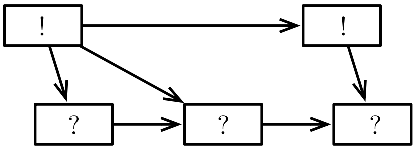

Figure 13. A stuttering channel

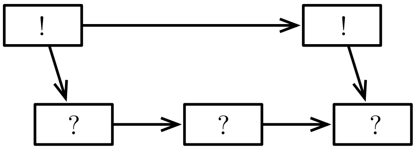

Figure 14. A fraudulent channel

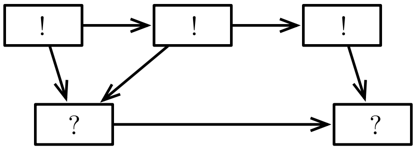

Figure 15. A confusing channel

Definition 9.1 (Non-lossy channel)**.** For a non-lossy channel:

- $! \rightarrow ?$ is a total relation on outputs
- $\mathbb{U}\,? \leftarrow ! \;=\; \mathbb{U}\,!$
- $! \;\subseteq\; ! \rightarrow ? \leftarrow !$
- $\forall e \in !.\, \exists f \in ?.\, e \rightarrow f$

A stuttering channel is one in which the same output is read more than once, as shown in Fig. 13. A non-stuttering channel is defined as one for which each output is read at most once. In other words, the data flow from an output to the corresponding input is a function.

Definition 9.2 (Non-stuttering channel)**.** For a non-stuttering channel:

- $! \rightarrow ?$ is a partial function
- $? \leftarrow ! \rightarrow ? \;\subseteq\; ?$
- $\forall e \in !.\, \forall f_1, f_2 \in ?.\, e \rightarrow f_1 \wedge e \rightarrow f_2 \Rightarrow f_1 = f_2$

The 'stuttering' pattern also describes the concept of a 'de-marshalling' or 'unpacking' channel. Each output is, say, a string, but each input is a separate character taken successively from that string, and delivered to a sequence of input operations. Unpacking is similar in its synchronisation behaviour to message duplication. It can be distinguished by the value labels on the arrows; also, stuttering is usually accidental and non-deterministic (as well as being undesirable). This is revealed by the fact that any channel which has a stuttering trace also has a similar trace with the stutter removed.

A fraudulent channel is one which may deliver a message that was never sent, as shown in Fig. 14. There are input events that are not the target of any send arrow.

Definition 9.3 (Non-fraudulent channel)**.** For a non-fraudulent channel:

- $! \rightarrow ?$ is surjective on inputs
- $\mathbb{U}\,! \rightarrow ? \;=\; \mathbb{U}\,?$
- $? \;\subseteq\; ? \leftarrow ! \rightarrow ?$
- $\forall e \in ?.\, \exists f \in !.\, f \rightarrow e$

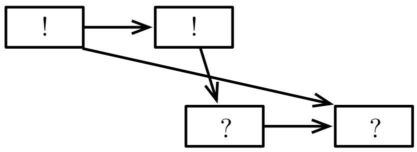

Figure 16. An overtaking channel

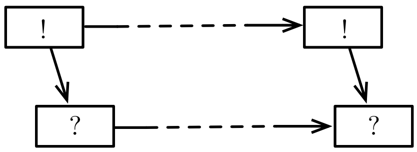

Figure 17. A non-overtaking channel

A confusing channel is one that delivers the result of several outputs to the same input, as shown in Fig. 15. Fraudulent and confusing channels are very similar to lossy and stuttering channels, except that the fault is manifested by the inputs rather than the outputs. Specification of the absence of these faults is also very similar to what we have described already in the case of outputs, and we will not make them explicit.

Definition 9.4 (Non-confusing channel)**.** For a non-confusing channel:

- $! \rightarrow\ ?$ is injective on outputs
- $! \rightarrow\ ? \leftarrow\ ! \ \subseteq\ !$
- $\forall e_1, e_2 \in !. \forall f \in ?. e_1 \rightarrow f \wedge e_2 \rightarrow f \Rightarrow e_1 = e_2$

The 'confusing' pattern also describes a 'packing' or 'marshalling' channel. The output events present a long string of characters, whereas the inputting instructions get fewer, larger blocks of characters, say sentences.

Finally, we consider an overtaking channel, which allows messages to be input in an order different to that which was output, as shown in Fig. 16. Overtaking is characteristic of wide-area store-and-forward networks like the internet. It results when successive messages take different paths through the network and the message sent earlier takes a longer or slower path than the message sent later.

Overtaking can be forbidden by a rule stating that the order of input of the messages is the same as the order that they were output, and vice-versa. In other words, the data flow relation from output to input is monotonic (order-preserving), and so is its converse, as exhibited in Fig. 17. Two-way monotonicity of a relation is expressed by a commuting equation as follows:

Definition 9.5 (Non-overtaking channel)**.** For a non-overtaking channel:

$$(! \rightarrow)^* ! \rightarrow ? \ = \ ! \rightarrow (? \rightarrow)^* ?$$

10. Threads

The threads of a program can also be regarded as a kind of resource. A thread's allocation event is triggered as a result of a 'fork' event by some other thread, and its disposal event must occur before it can be 'joined' by another thread. All the events resulting from execution of the thread must come in between its allocation and disposal events. Figure 18 shows the dependencies that result from this pattern of behaviour. For simplicity, we have drawn each thread as a sequential resource, but in principle each could contain internal concurrency.

The execution of a thread comprises of the execution of its atomic commands. As an example of an atomic command, Fig. 19 takes the addition of the value of the variable y

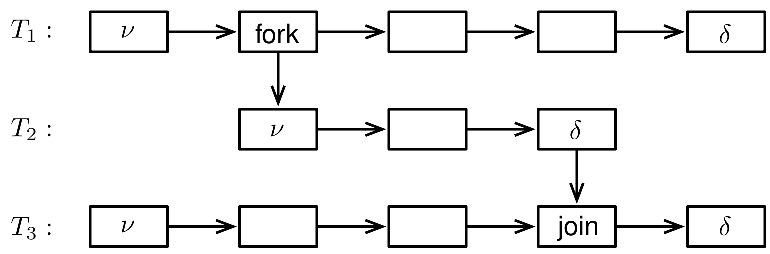

Figure 18. Threads

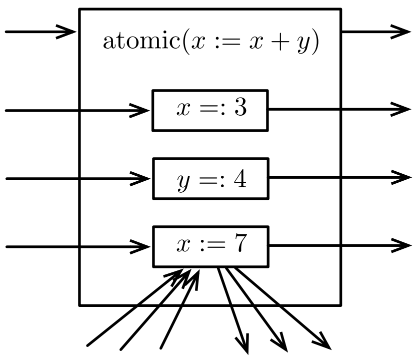

Figure 19. An atomic assignment

to the variable x. It is a common practice to signal atomicity of commands by applying a function called 'atomic'. As usual, atomicity of execution of the atomic command is a single event, pictured as the outer box in the figure. The type of the command, $x := x+y$, is written inside it. Inside the same box, we also write the finer-grained atomic events which occur as part of the coarser-grained atomic event. They represent the boxes that occur in the data flow diagrams for the individual resources x and y that participate in this action. In this example, they are shown as a fetch of the value 3 from x, the fetch of the value 4 from y, and the assignment of their sum 7 to x.

The two arrows at the top of Fig. 19 are control arrows connecting the atomic event into the trace of the thread that invoked it. Below them are the data flow arrows for each of the component events of the atomic assignment. The arrows of the topmost internal box are the fan-out and fan-in arrows between consecutive assignments and fetches of the variable x; just below it, there are similar arrows for y. The pair of horizontal arrows at the bottom are part of the assignment chain for x. The bottom six arrows are fan-out and fan-in arrows between this assignment and other fetches of the same variable x. The diagram is slightly misleading, in that one of the dependencies has been drawn twice, as two separate arrows. The arrow leaving the fetch of x and the arrow entering the assignment to x are, in fact, just the two ends of the same arrow.

The inclusion of the smaller boxes inside the larger box is indicative of the nesting of atomic regions at various levels of granularity. More formally, this can be represented as a function from finer grain events to coarser grain events. The function is sometimes written with the name 'atomic' in the program itself, using the syntax shown in Fig. 19.

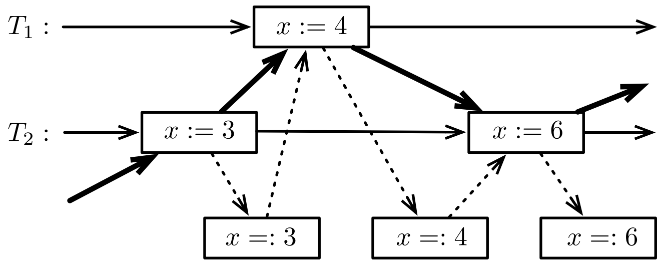

Figure 20. A shared variable

There are many ways of implementing atomicity, for example by inhibiting interrupts, or by acquiring and releasing semaphores, or by transactional protocols. Sometimes, responsibility for atomicity is undertaken by the programmer. All methods of implementation should achieve the effect modelled by our pictures. There is much more to be said about atomicity and the possible constraints on its use, but we shall not do so here.

11. Shared variables and weakly-consistent memory

Let us now explore in greater detail the interaction between a shared variable and the threads that share it. Figure 20 shows a fragment from a trace of a variable named x, in which we have differentiated three kinds of arrow. The three assignments to x are connected by the chain of bold arrows. The fetches are connected to adjacent assignments by dotted arrows. The bold and dotted arrows apply to private variables, which are used exclusively by a single thread. They apply equally to variables that are shared between many threads. We will concentrate on the shared case in the following discussion.

The two chains of horizontal, solid arrows belong to two threads that share this resource. The diagram shows how thread T_1 interferes with T_2 by assigning the value 4 to the shared variable; in between, two other assignments are made by T_2. Thread T_2 knows nothing about the assignment of 4 to x when it occurs, but it does notice the resulting interference as a spontaneous change in the value of x (from value 3 to value 4), which occurs between its own two assignments to the variable.

Figure 20 demonstrates how simply our model deals with the behaviour of shared variables. There was no need for any extra definitions or theorems. The shared variables turn out to be the simple and general case. It is the private variable that needs to be defined as a special case, by placing the obvious restrictions on the way in which the variable is used. As a result, all of the theorems proved of our general shared variable remain true for all of the more complex restricted cases. This kind of re-use of previously proved theorems is exactly what motivates our search for unifying theories.

Unfortunately, the simplicity of our definition of a variable is not matched by the simplicity of writing a program that uses shared variables. And the complexity of assuring its correctness is certainly far greater.

Even more unfortunately, modern multi-core processor architecture makes both the definition and the programming of shared variables even more complicated. Nevertheless, our concept of data flow is capable of dealing with the added complexity, as we will

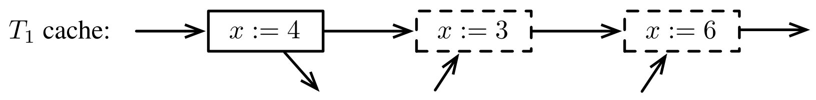

Figure 21. A cache

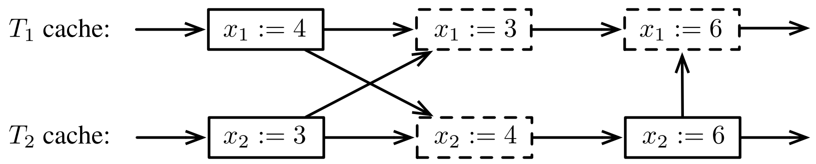

Figure 22. Two caches

show in the remainder of this article. We will redefine the behaviour of a shared variable in a way that models the weakness of weakly consistent memory. However, we will deal only with the simplest possible cases, and ignore the question of whether any current multi-core architecture actually behaves as badly as we suggest.

11.1. Modelling caches

In a modern multi-core architecture, each separate core has its own cache memory. The primary purpose of the cache memory is to reduce the average latency of global memory access by that core. We will disregard the concern for efficiency, and model only the logical effect of the cache on the behaviour of the variable and the thread which uses it. In principle, each cache simulates the content of the entire shared global memory, as seen by the core that owns the cache. Any assignment made by that core has an immediate effect on the content of its own cache, and therefore on its own view of global memory.

In Fig. 21, we introduce the convention that an assignment made by the owning thread has a solid border, whereas the (interfering) assignments made by other threads have a dashed border. Some time after an assignment is made by a thread, the information about its occurrence will be propagated automatically to the caches of other threads, as shown by the arrow leading from the first assignment in Fig. 21. Similarly, assignments made by other threads will eventually be propagated to the cache of this thread, as shown by the arrows leading to the other two assignments in the figure.

Figure 22 adds the behaviour of the cache of another thread, with the variable name subscripted to avoid confusion. The arrows show the communication of values between the two caches. This is a buffered communication, with an arbitrary delay between a local assignment and its effect on the cache of another thread. In a hardware implementation, the assignment is effectively buffered in a hardware write queue that is local to each thread. As a result, the value 3, assigned by T_2, arrives at T_1 after the latter's local assignment of value 4. But news of the assignment of 4 arrives at T_2's cache after it has assigned 3. Thus the two threads see the effect of these two assignments in a different order. This is the essence of the weakly consistent memory of modern multi-core processors. However, there are many variations, nearly always stronger than the weak case which we have shown.

In Fig. 23, we have added to each thread a row of fetches made by that thread from its own local memory. Comparison of the two rows reveals the inconsistency of the ordering

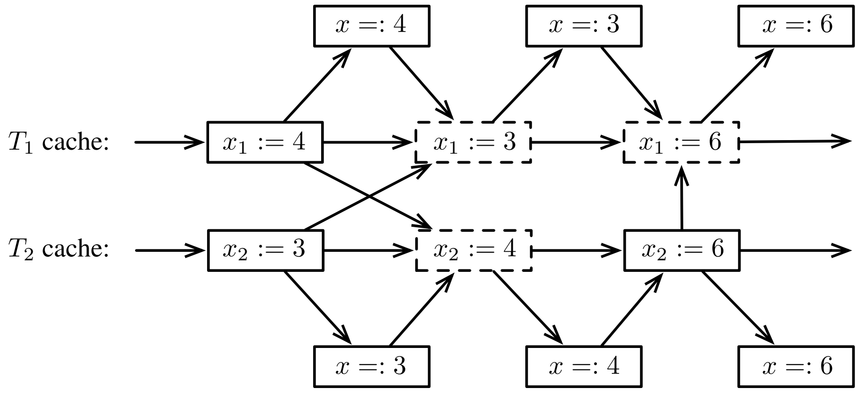

Figure 23. Partial store ordering

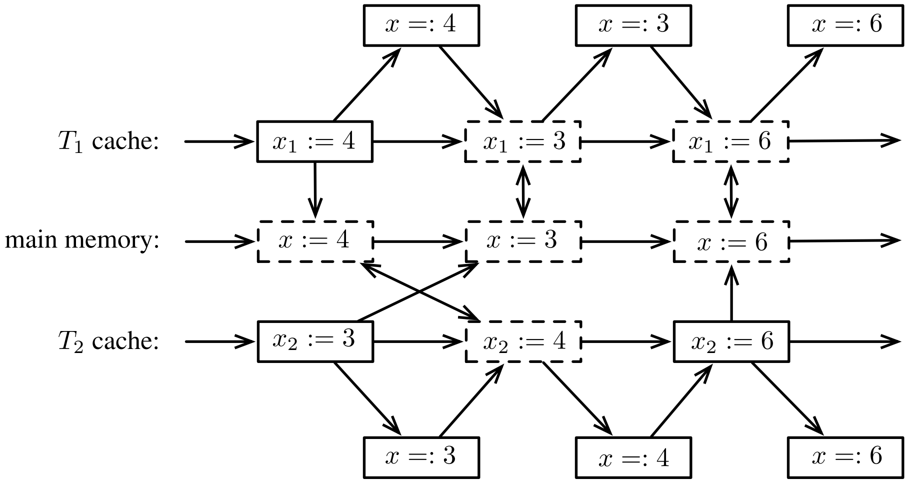

Figure 24. Total store ordering

of the first two assigned values as seen by each thread. When more threads are added, every thread could see a different interleaving of the assignments made by all the other threads. The degree of non-determinacy is daunting.

That is why many multi-core architectures adopt a stronger memory model than the one illustrated in Fig. 23. This stronger model, shown in Fig. 24, has an additional component representing the hardware of the real main memory shared by all the threads. This component records every write-back to main memory from the caches of every thread. Each cache sends its assignments directly to the main memory, rather than to each other. When the main memory component writes the assigned value, it simultaneously sends that assignment, synchronously, to the caches of all the threads, except the one that originally made the assignment. This is indicated by the double-ended arrows in Fig. 24.

The synchrony ensures that there is single canonical ordering to assignments actually made to the main store, and that this same ordering is seen by all the threads – with the exception of assignments made by the thread itself, which are seen too early.

This stronger memory model is known as total store ordering. Upon comparison with the model depicted in Fig. 23, we can see that the introduction of the main memory makes no difference if there are only two threads. In fact, four threads are needed to reveal the true difference between the two models.

The synchronous propagation of an assignment to a large number of caches could be very inefficient; so it is usually implemented in an indirect way, by invalidating every hardware cache line which holds the assigned variable. If and when the thread needs access to the variable x, it is thereby forced to go to the real shared memory to get it. The effect however is the same as we have described – at least at the level of granularity modelled by our diagrams.

This section has been a brief and over-simplified introduction to the ideas of weakly-consistent memory. It diverges from the model actually implemented in current multi-core architectures in many ways. Nevertheless, the explanation is illustrative of the power of data flow to describe important computational ideas, and the power of the relational calculus to reason about them.

12. Conclusion

Data flow is a primitive concept, adequate to describe the dynamic behaviour of many kinds of computing resource. Relational calculus, illustrated by labelled graphs, provides a general framework adequate for a unifying theory of data flow.

Acknowledgements

Many thanks to colleagues who have contributed detailed suggestions to improve the substance and presentation of this work, and in particular: Peter Höfner, Jay Misra, Bernhard Möller, Jørgen Steensgaard and Viktor Vafeiadis.

References

[1] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10), 1969.

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

[3] R.D. Maddux. *Relation Algebras*, volume 150 of *Studies in Logic and the Foundations of Mathematics*, pages 1–33. Elsevier, 2006.

[4] Ian Wehrman, C.A.R. Hoare, and Peter W. O'Hearn. Graphical models of separation logic. *Information Processing Letters*, 109(17):1001 – 1004, 2009.

[5] Glynn Winskel and Mogens Nielsen. Models for concurrency. In *Handbook of Logic in Computer Science*, volume 4, pages 1–148. Oxford University Press, Oxford, UK, 1995.