Electronic Notes in Theoretical Computer Science 45 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume45.html 19 pages

Transfer Principles for Reasoning About Concurrent Programs

Stephen Brookes[1]

Department of Computer Science
Carnegie Mellon University
Pittsburgh, USA

Abstract

In previous work we have developed a *transition trace* semantic framework, suitable for shared-memory parallel programs and asynchronously communicating processes, and abstract enough to support compositional reasoning about safety and liveness properties. We now use this framework to formalize and generalize some techniques used in the literature to facilitate such reasoning. We identify a *sequential-to-parallel transfer theorem* which, when applicable, allows us to replace a piece of a parallel program with another code fragment which is *sequentially* equivalent, with the guarantee that the safety and liveness properties of the overall program are unaffected. Two code fragments are said to be sequentially equivalent if they satisfy the same partial and total correctness properties. We also specify both coarse-grained and fine-grained version of trace semantics, assuming different degrees of atomicity, and we provide a *coarse-to-fine-grained transfer theorem* which, when applicable, allows replacement of a code fragment by another fragment which is *coarsely* equivalent, with the guarantee that the safety and liveness properties of the overall program are unaffected even if we assume fine-grained atomicity. Both of these results permit the use of a simpler, more abstract semantics, together with a notion of semantic equivalence which is easier to establish, to facilitate reasoning about the behavior of a parallel system which would normally require the use of a more sophisticated semantic model.

1 Introduction

It is well known that syntax-directed reasoning about behavioral properties of parallel programs tends to be complicated by the combinatorial explosion

[1] This research is sponsored in part by the National Science Foundation (NSF) under Grant No. CCR-9988551. The views and conclusions contained in this document are those of the author, and should not be interpreted as representing the official policies, either expressed or implied, of the NSF or the U.S. government.
[2] Email: brookes@cs.cmu.edu

inherent in keeping track of dynamic interactions between code fragments. Simple proof methodologies based on state-transformation semantics, such as Hoare-style logic, do not adapt easily to the parallel setting, because they abstract away from interaction and only retain information about the initial and final states observed in a computation. A more sophisticated semantic model is required, in which an accurate account can be given of interaction.

Trace semantics provides a mathematical framework in which such reasoning may be carried out [2,3,4,5]. The trace set of a program describes all possible patterns of interaction between the program and its "environment", assuming fair execution [9]. One can define both a *coarse-grained* trace semantics, in which assignment and boolean expression evaluation are assumed to be executed atomically, and a *fine-grained* trace semantics, in which reads and writes (to shared variables) are assumed to be atomic. Trace semantics can be defined denotationally, and is *fully abstract* with respect to a notion of program behavior which subsumes partial correctness, total correctness, safety properties, and liveness properties [2].

To some extent program proofs may be facilitated by a number of laws of program equivalence, validated by trace semantics, which allow us to deduce properties of a program by analyzing instead a semantically equivalent program with simpler structure. The use of a succinct and compact notation for trace sets (based on extended regular expressions) can also help streamline program analysis. Yet the problem remains that in general the trace set of a program can be difficult to manipulate and hard to use to establish correctness properties. Trace sets tend to be rather complex mathematical objects, since a trace set describes *all* possible interactions between the program and *any* potential environment. For the same reason, both the coarse- and the fine-grained trace semantics induce a rather discriminating notion of semantic equivalence, and few laws of equivalence familiar from the sequential setting also hold in all parallel contexts. It can therefore be difficult to establish trace equivalence of programs merely by direct manipulation of the semantic definitions, or by using trace-theoretic laws of program equivalence in a syntax-directed manner.

In practice, parallel systems ought to be designed carefully to ensure that the interactions between component processes are highly disciplined and constrained. Moreover, when analyzing the properties of code to be run in tightly controlled contexts, we ought to be able to work within a simpler semantic model (or, at least, within a reduced subset of the trace semantics) whose simplicity reflects this discipline. Correspondingly, whenever we know that a program fragment will be used in a limited form of context, we would like to be able to employ forms of reasoning which take advantage of the limitations.

For example, we might know that a piece of code is going to be used "sequentially" inside a parallel program (in a manner to be made precise soon) and want to use Hoare-style reasoning about this code in establishing safety and liveness properties of the whole program. It is not generally safe to do so, since laws of program equivalence that hold in the sequential setting cease to be valid in parallel languages because of the potential for interference between concurrently executing code. Yet *local variables* can only be accessed by processes occurring within a syntactically prescribed scope, and cannot be changed by any other processes running concurrently, so we ought to be able to take advantage of this non-interference property to simplify reasoning about code which only affects local variables. In particular, when local variables are only ever used sequentially, in a context whose syntactic structure guarantees that no more than one process ever gains concurrent access, we should be able to employ Hoare-style reasoning familiar from the sequential setting. We would like to know the extent to which this idea can be made precise, and when this technique is applicable.

In a similar vein, it is usually regarded as realistic to assume fine-grained atomicity when trying to reason about program behavior, but more convenient to make the less realistic but simplifying assumption of coarse granularity, since this assumption may help to reduce the combinatorial explosion. We would like to be able to identify conditions under which it is safe to do so.

A number of *ad hoc* techniques have been proposed along these lines in the literature, usually without detailed consideration of semantic foundations [1]. Their common aim is to facilitate concurrent program analysis by allowing replacement of a code fragment by another piece of code with "simpler" behavioral properties that permit an easier correctness proof.

In this paper we use the trace-theoretic framework to formalize and generalize some of these techniques. By paying careful attention to the underlying semantic framework we are able to recast these techniques in a more precise manner and we can be more explicit about the (syntactic and semantic) assumptions upon which their validity rests. Since these techniques allow us to deduce program equivalence properties based on one semantic model by means of reasoning carried out on top of a different semantic model, we refer to our results as *transfer principles*. We provide transfer principles specifically designed to address the two example scenarios used for motivation above: a *sequential-to-parallel* transfer principle allowing use of Hoare-style reasoning, and a *coarse-to-fine* transfer principle governing the use of coarse semantics in fine-grained proofs of correctness.

Our work can be seen as further progress towards a theory of *context-sensitive development of parallel programs*, building on earlier work of Cliff Jones [8] and spurred on by the recent Ph. D. thesis of Jürgen Dingel [7]. We focus our attention initially on some methodological ideas presented in Greg Andrews's book on concurrent programming [1]. Later we intend to explore more fully the potential of our framework as a basis for further generalization and to extend our results to cover some of the contextual refinement ideas introduced by Dingel.

In this preliminary version of the paper we omit explicit details of the underlying trace semantics, which the reader can find in [2], and we omit most of the proofs, which require detailed use of the semantic definitions.

2 Syntax

2.1 The programming language

Our parallel programming language is described by the following abstract grammar for commands c, in which b ranges over boolean-valued expressions, e over integer-valued expressions, x over identifiers, a over atomic commands (finite sequences of assignments), and d over declarations. The syntax for expressions is conventional and is assumed to include the usual primitives for arithmetic and boolean operations.

$$\begin{aligned} c ::= {} & \textbf{skip} \mid x{:=}e \mid c_1; c_2 \mid \textbf{if } b \textbf{ then } c_1 \textbf{ else } c_2 \mid \\ & \textbf{while } b \textbf{ do } c \mid \textbf{local } d \textbf{ in } c \mid \\ & \textbf{await } b \textbf{ then } a \mid \ c_1 \| c_2 \\ d ::= {} & x = e \mid d_1; d_2 \\ a ::= {} & \textbf{skip} \mid x{:=}e \mid a_1; a_2 \end{aligned}$$

A command of form **await** b **then** a is a *conditional atomic action*, and causes the execution of a without interruption when executed in a state satisfying the test expression b; when executed in a state in which b is false the command idles.

A *sequential program* is just a command containing no **await** and no parallel composition.

Assume given the standard definitions of `free`(e) and `free`(b), the set of identifiers occurring free in an expression. We will use the standard definitions of `free`(c) and `free`(d) for the sets of identifiers occurring free in a command or a declaration, and `dec`(d), the set of identifiers declared by d.

2.2 Parallel, atomic, and sequential contexts

A *context* is a command which may contain a syntactic "hole" (denoted $[-]$) suitable for insertion of another command. Formally, the set of (parallel) contexts, ranged over by C, is described by the following abstract grammar, in which c_1, c_2 again range over commands:

$$\begin{aligned} C ::= {} & [-] \mid \textbf{skip} \mid x{:=}e \mid C; c_2 \mid c_1; C \mid \\ & \textbf{if } b \textbf{ then } C \textbf{ else } c_2 \mid \textbf{if } b \textbf{ then } c_1 \textbf{ else } C \mid \\ & \textbf{while } b \textbf{ do } C \mid \textbf{local } d \textbf{ in } C \mid \\ & \textbf{await } b \textbf{ then } a \mid \\ & C \| c_2 \mid c_1 \| C \end{aligned}$$

Note that our abstract grammar for contexts only allows at most one hole to appear in any particular context. It would be straightforward to adopt a more general notion of multi-holed context, but the technical details would become more involved and in any case there is no significant loss of generality.

We also introduce the notion of an *atomic context*, i.e. a parallel context whose hole occurs inside the body of an **await** command. We will use A to range over atomic contexts.

A *sequential context* is a limited form of context in which the hole never appears in parallel. We can characterize the set of sequential contexts, ranged over by S, as follows:

$$\begin{aligned} S ::= {} & [-] \mid \mathbf{skip} \mid x{:=}e \mid S; c_2 \mid c_1; S \mid \\ & \mathbf{if}\ b\ \mathbf{then}\ S\ \mathbf{else}\ c_2 \mid \mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ S \mid \\ & \mathbf{while}\ b\ \mathbf{do}\ S \mid \mathbf{local}\ d\ \mathbf{in}\ S \mid \\ & \mathbf{await}\ b\ \mathbf{then}\ a \mid\ c_1 \| c_2 \end{aligned}$$

The important point in this definition is that $c_1 \| S$ is not a sequential context even when S is sequential, but we do allow "harmless" uses of parallelism inside sequential contexts, as for example in $(c_1 \| c_2); [-]$. The key feature is that sequentiality of S ensures that when we fill the hole with a command we have the guarantee that the command will not be executed concurrently with any of the rest of the code in S.

We write $C[c]$ for the command obtained by inserting c into the hole of C. We use similar notation $A[a]$ for the result of inserting an atomic command a into an atomic context A, and $S[c]$ for the result of inserting a (parallel) command c into a sequential context S.

It is easy to define the set $\mathtt{free}(C)$ of identifiers occurring free in a context C, as usual by structural induction. Similarly we let $\mathtt{free}(S)$ and $\mathtt{free}(A)$ be the sets of identifiers occurring free in sequential context S and in atomic context A.

Contexts may also have a *binding* effect, since the hole in a context may occur inside the scope of one or more (nested) declarations, and free occurrences of identifiers in a code fragment may become bound after insertion into the hole. For example, the context

$$\mathbf{local}\ y = 0\ \mathbf{in}\ ([-] \| y{:=}z + 1)$$

binds y, but not z. On the other hand, the context

$$(\mathbf{local}\ y = 0\ \mathbf{in}\ c_1) \| ([-]; c_2)$$

does not bind any identifier, since the hole does not occur inside a subcommand of **local** form.

To be precise about this possibility we make the following definition. We also make use of analogous notions for sequential contexts and for atomic contexts, which may be defined in the obvious analogous way. Although we will not prove this here, it follows from the definition that (except for the case of a degenerate context with no hole) for all contexts C and commands c, $\mathtt{free}(C[c]) = \mathtt{free}(C) \cup (\mathtt{free}(c) - \mathtt{bound}(C))$.

Definition 2.1 For a context C, let $\mathtt{bound}(C)$ be the set of identifiers for which there is a binding declaration enclosing the hole in C, defined as follows:

$$\mathtt{bound}([-]) = \emptyset$$

$$\mathtt{bound}(x{:=}e) = \emptyset$$

$$\mathtt{bound}(C; c_2) = \mathtt{bound}(c_1; C) = \mathtt{bound}(C)$$

$$\mathtt{bound}(\mathbf{if}\ b\ \mathbf{then}\ C\ \mathbf{else}\ c_2) = \mathtt{bound}(\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ C) = \mathtt{bound}(C)$$

$$\mathtt{bound}(\mathbf{while}\ b\ \mathbf{do}\ C) = \mathtt{bound}(C)$$

$$\mathtt{bound}(\mathbf{await}\ b\ \mathbf{then}\ a) = \emptyset$$

$$\mathtt{bound}(C \| c_2) = \mathtt{bound}(c_1 \| C) = \mathtt{bound}(C)$$

$$\mathtt{bound}(\mathbf{local}\ d\ \mathbf{in}\ C) = \mathtt{bound}(C) \cup \mathtt{dec}(d)$$

3 Semantics

3.1 Operational semantics

We assume conventional coarse-grained and fine-grained operational semantics for expressions and commands [2]. In both cases command configurations have the form $\langle c, s\rangle$, where c is a command and s is a state. A state s determines a (finite, partial) function from identifiers to variables, and a "store" mapping variables to their "current" integer values. We let $\mathbf{S}$ be the set of states. A transition of form

$$\langle c, s\rangle \rightarrow \langle c', s'\rangle$$

represents the effect of c performing an *atomic step* enabled in state s, resulting in a change of state to s', with c' remaining to be executed. A terminal configuration, in which all parallel component commands have terminated, is represented by a (final) state s. In a fine-grained semantics reads and writes to variables are atomic, but assignments and boolean condition evaluations need not be. In a coarse-grained semantics, assignments and boolean expressions are atomic.

A *computation* of a command c is a finite sequence of transitions, ending in a terminal configuration, or an infinite sequence of transitions that is *fair* to all parallel component commands of c. (We may also refer to a *fine-grained computation* or a *coarse-grained computation*, when we need to be precise about which granularity assumption is relevant.) We write $\langle c, s\rangle \rightarrow^* \langle c', s'\rangle$ to indicate a finite, possibly empty, sequence of transitions; and $\langle c, s\rangle \rightarrow^\omega$ to indicate the existence of a (weakly) fair infinite computation starting from a given configuration. An *interactive computation* is a finite or infinite sequence of transitions in which the state may be changed between steps, representing the effect of other commands executing in parallel. There is an analogous notion of fairness for interactive computations. A computation is just an interference-free interactive computation, that is, an interactive computation in which no external changes occur.

3.2 State-transformation semantics and sequential equivalence

Definition 3.1 The standard state-transformation semantics for programs, denoted $\mathcal{M}$, is characterized operationally by:

$$\mathcal{M}(c) = \{(s, s') \mid \langle c, s\rangle \to^* s'\} \;\cup\; \{(s, \bot) \mid \langle c, s\rangle \to^\omega\}.$$

Definition 3.2 Two programs c_1 and c_2 are *sequentially equivalent*, written $c_1 \equiv_{\mathcal{M}} c_2$, if and only if $\mathcal{M}(c_1) = \mathcal{M}(c_2)$.

As is well known, sequential equivalence is a congruence with respect to the sequential subset of our programming language. In fact, for all parallel programs c_1 and c_2, and all *sequential* contexts S,

$$c_1 \equiv_{\mathcal{M}} c_2 \;\Leftrightarrow\; S[c_1] \equiv_{\mathcal{M}} S[c_2].$$

However, the analogous property fails to hold for *parallel* contexts, because, for example, we have:

$$x{:=}x + 2 \equiv_{\mathcal{M}} x{:=}x + 1;\; x{:=}x + 1$$

but

$$x{:=}x + 2 \| y{:=}x \not\equiv_{\mathcal{M}} (x{:=}x + 1;\; x{:=}x + 1) \| y{:=}x.$$

3.3 Trace semantics

A transition trace of a program c is a finite or infinite sequence of steps, each step being a pair of states that represents the effect of a finite sequence of atomic actions performed by the program. A particular trace $(s_0, s'_0)(s_1, s'_1)\ldots(s_n, s'_n)\ldots$ of c represents a possible fair interactive computation of c in which the inter-step state changes (from s'_0 to s_1, and so on) are assumed to be caused by processes executing concurrently to c. Traces are "complete", representing an entire interactive computation, rather than "partial" or "incomplete". A trace is *interference-free* if the state never changes between successive steps along the trace, i.e. in the notation used above when we have $s'_i = s_{i+1}$ for all i. An interference-free trace represents a sequence of snapshots of the state taken during an interference-free fair computation.

Again we obtain both a coarse-grained notion of trace, based on the coarse interpretation of atomicity and the coarse-grained operational semantics, and a fine-grained notion of trace, based on the fine interpretation of atomicity and the fine-grained opeartional semantics. Both coarse- and fine-grained trace semantics interpret conditional atomic actions **await** b **then** a as atomic. The coarse-grained trace semantics, which we will denote $\mathcal{T}_{coarse}$, also assumes that assignments and boolean expression evaluations are atomic. The fine-grained semantics, denoted $\mathcal{T}_{fine}$, assumes only that reads and writes to simple variables are atomic. In the rest of this paper, when stating a result which holds for both fine- and coarse-grained semantics, we may use $\mathcal{T}$ to stand for either version of the trace semantic function.

Trace semantics can be defined compositionally, and we note in particular that the traces of $c_1 \| c_2$ are obtained by forming fair merges of a trace of c_1 with a trace of c_2, and the traces of $c_1; c_2$ are obtained by concatenating a trace of c_1 with a trace of c_2, closing up under stuttering and mumbling as required. The traces of **local** $x = e$ **in** c do not change the value of (the "global" version of) x, and are obtained by projection from traces of c in which the value of (the "local" version of) x is never altered between steps.

A parallel program denotes a trace set closed under two natural conditions termed *stuttering* and *mumbling*, which correspond to our use of a step to represent finite sequences of actions: idle or stuttering steps of form (s, s) may be inserted into traces, and whenever two adjacent steps $(s, s')(s', s'')$ share the same intermediate state they can be combined to produce a mumbled trace which instead contains the step (s, s''). The closure properties ensure that trace semantics is *fully abstract* with respect to a notion of behavior which assumes that we can observe the state during execution. As a result trace semantics supports compositional reasoning about safety and liveness properties. Safety properties typically assert that no "bad" state ever occurs when a process is executed, without interference, from an initial state satisfying some pre-condition. A liveness property typically asserts that some "good" state eventually occurs. When two processes have the same trace sets it follows that they satisfy identical sets of safety and liveness properties, in all parallel contexts.

3.4 Fine- and coarse-grained semantic equivalences

When using coarse-grained semantics one can safely use algebraic laws of arithmetic to simplify reasoning about program behavior. For instance, in coarse-grained trace semantics the assignments $x{:=}x + x$ and $x{:=}2 \times x$ are equivalent. This feature can be used to considerable advantage in program analysis. However, coarse granularity is in general an unrealistic assumption since implementations of parallel programming languages do not generally guarantee that assignments are indeed executed indivisibly.

The fine-grained trace semantics is closer in practice to conventional implementations, but less convenient in program analysis. When using fine-grained semantics one cannot assume with impunity that algebraic laws of expression equivalence remain valid. For instance, the assignments $x{:=}x+x$ and $x{:=}2\times x$ are not equivalent in fine-grained trace semantics; this reflects the fact that the former reads x twice, so that if x is changed during execution (say from 0 to 1), the value assigned may be $0, 1$ or 2, whereas the latter assignment (under the same circumstances) would assign either 0 or 2.

It should be clear from the above discussion, even without seeing all of the semantic definitions, that despite the connotations suggested by our use of "fine" *vs.* "coarse", these two trace semantic variants induce *incomparable* notions of semantic equivalence. Let us write

$$c_1 \equiv_{\mathit{coarse}} c_2 \;\Leftrightarrow\; \mathcal{T}_{\mathit{coarse}}(c_1) = \mathcal{T}_{\mathit{coarse}}(c_2)$$

$$c_1 \equiv_{\mathit{fine}} c_2 \;\Leftrightarrow\; \mathcal{T}_{\mathit{fine}}(c_1) = \mathcal{T}_{\mathit{fine}}(c_2)$$

For instance, we have already seen a pair of programs which are equivalent in coarse-grained semantics but not in fine-grained:

$$x{:=}x + x \equiv_{\mathit{coarse}} x{:=}2 \times x, \quad x{:=}x + x \not\equiv_{\mathit{fine}} x{:=}2 \times x,$$

so that $c_1 \equiv_{\mathit{coarse}} c_2$ does not always imply $c_1 \equiv_{\mathit{fine}} c_2$.

The converse implication also fails, as shown by the programs $x{:=}x + x$ and

$$\textbf{local}\ y = 0; z = 0\ \textbf{in}\ (y{:=}x; z{:=}x; x{:=}y + z)$$

These are equivalent in fine-grained but not in coarse-grained semantics.

Despite the incomparability of fine-grained equivalence and coarse-grained equivalence, for any particular program c the coarse-grained trace set will be a subset of its fine-grained traces:

$$\mathcal{T}_{\mathit{coarse}}(c) \subseteq \mathcal{T}_{\mathit{fine}}(c),$$

so that it is reasonable to refer to the coarse-grained semantics as "simpler".

We also remark that the state-transformation semantics of a parallel program is determined by its trace semantics, in fact by its *interference-free* traces, since $(s, s') \in \mathcal{M}(c)$ if and only if $(s, s') \in \mathcal{T}_{\mathit{fine}}(c)$, and $(s, \perp) \in \mathcal{M}(c)$ if and only if there is an infinite interference-free trace in $\mathcal{T}_{\mathit{fine}}(c)$ beginning from state s. (Here we adopt the usual pun of viewing (s, s') simultaneously as a *pair* belonging to $\mathcal{M}(c)$ and as a *trace* of length 1 belonging to $\mathcal{T}(c)$. Such a trace is trivially interference-free.)

Each trace equivalence is a congruence for the entire parallel language, so that for all contexts C and parallel commands c_1 and c_2 we have:

$$c_1 \equiv_{\mathit{coarse}} c_2 \;\Leftrightarrow\; C[c_1] \equiv_{\mathit{coarse}} C[c_2]$$

$$c_1 \equiv_{\mathit{fine}} c_2 \;\Leftrightarrow\; C[c_1] \equiv_{\mathit{fine}} C[c_2]$$

Moreover, $c_1 \equiv_{\mathit{fine}} c_2$ implies $c_1 \equiv_{\mathcal{M}} c_2$, but the converse implication is not generally valid.

4 Reads and writes of a command

To prepare the ground for our transfer principles, we first need to define for each parallel program c the *multiset* $\texttt{reads}(c)$ of identifier occurrences which appear free in non-atomic sub-expressions of c. It is vital here, as suggested by the terminology, to keep track of how many references the program makes, to each identifier. We need only be concerned with non-atomic subphrases, since these are the only ones whose execution may be affected by concurrent activity.

We also need to refer to the analogous notions for expressions and for declarations; since we have not provided a full grammar for expressions we will give details only for a few key cases, which suffice for understanding all of the examples which follow and which convey the general ideas.

For precise mathematical purposes, we may think of a multiset as a set of identifiers equipped with a non-negative multiplicity count. In the empty multiset every identifier has multiplicity 0. When M_1 and M_2 are multisets, we let $M_1 \cup_+ M_2$ be the multiset union in which multiplicities are added, and $M_1 \cup_{max} M_2$ be the multiset union in which multiplicities are combined using max. That is, an identifier x which occurs n_1 times in M_1 and n_2 times in M_2 will occur $n_1 + n_2$ times in $M_1 \cup_+ M_2$ and $max(n_1, n_2)$ times in $M_1 \cup_{max} M_2$.

We write $\{\!|x|\!\}$ for the singleton multiset containing a single occurrence of x. We also write $\{\!|\ |\!\}$ for the empty multiset. The cardinality of a multiset M is denoted $|M|$.

Each version of union is symmetric and associative:

$$M_1 \cup_+ M_2 = M_2 \cup_+ M_1$$

$$M_1 \cup_+ (M_2 \cup_+ M_3) = (M_1 \cup_+ M_2) \cup_+ M_3$$

$$M_1 \cup_{max} M_2 = M_2 \cup_{max} M_1$$

$$M_1 \cup_{max} (M_2 \cup_{max} M_3) = (M_1 \cup_{max} M_2) \cup_{max} M_3$$

In addition, $\cup_{max}$ is idempotent:

$$M \cup_{max} M = M$$

Obviously $\cup_+$ is not idempotent.

The empty multiset is a unit for both forms of union, since

$$M \cup_+ \{\!|\ |\!\} = M \cup_{max} \{\!|\ |\!\} = M.$$

Given a multiset M and a set X of identifiers, we define $M - X$ to be the multiset obtained from M by removing all occurrences of identifiers in X, and we let $M \cap X$ be the multiset consisting of those members of M which are also in X, with the same multiplicities as they have in M.

We are now ready to define the read multiset of an expression. Again we include only a few representative cases. Note that we will use the additive form of multiset union for an expression of form $e_1 + e_2$ (and also, in general, for expressions built with binary operators), because we want to count the number of times an identifier needs to be read during the evaluation of an expression.

Definition 4.1 The multiset $\mathtt{reads}(e)$ of free identifier occurrences in an expression e is given inductively by:

$$\mathtt{reads}(n) = \{\!\mid\ \mid\!\}$$

$$\mathtt{reads}(x) = \{\!\mid x \mid\!\}$$

$$\mathtt{reads}(e_1 + e_2) = \mathtt{reads}(e_1) \cup_{+} \mathtt{reads}(e_2)$$

A similar definition can be given for boolean expressions.

Definition 4.2 The multiset $\mathtt{reads}(d)$ of free identifier occurrences in a declaration d is given inductively by:

$$\mathtt{reads}(x = e) = \mathtt{reads}(e)$$

$$\mathtt{reads}(d_1; d_2) = \mathtt{reads}(d_1) \cup_{max} (\mathtt{reads}(d_2) - \mathtt{dec}(d_1))$$

Here we combine using maximum since d may require the separate evaluation of several sub-expressions.

Now we can provide the definition for commands:

Definition 4.3 The multiset $\mathtt{reads}(c)$ of free identifier occurrences read by command c is given inductively by:

$$\mathtt{reads}(\mathbf{skip}) = \{\!\mid\ \mid\!\}$$

$$\mathtt{reads}(x{:=}e) = \mathtt{reads}(e)$$

$$\mathtt{reads}(c_1; c_2) = \mathtt{reads}(c_1 \| c_2) = \mathtt{reads}(c_1) \cup_{max} \mathtt{reads}(c_2)$$

$$\mathtt{reads}(\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2) = \mathtt{reads}(b) \cup_{max} (\mathtt{reads}(c_1) \cup_{max} \mathtt{reads}(c_2))$$

$$\mathtt{reads}(\mathbf{while}\ b\ \mathbf{do}\ c) = \mathtt{reads}(b) \cup_{max} \mathtt{reads}(c)$$

$$\mathtt{reads}(\mathbf{await}\ b\ \mathbf{then}\ a) = \{\!\mid\ \mid\!\}$$

$$\mathtt{reads}(\mathbf{local}\ d\ \mathbf{in}\ c) = \mathtt{reads}(d) \cup_{max} (\mathtt{reads}(c) - \mathtt{dec}(d))$$

Again we use the maximum-forming union operation to combine the counts from all sub-expression evaluations. Notice that we regard an **await** command as having *no* reads, because it will be executed atomically and its effect will therefore be immune from concurrent interference.

Next we define the *set* $\mathtt{writes}(c)$ of identifier occurrences which occur free in c as targets of assignments. It will turn out that we do not need an accurate count of how many times an individual identifier is assigned, just the knowledge of whether or not each identifier is assigned to: even once is bad enough. Our definition ensures that $x \in \mathtt{writes}(c)$ if and only if there is at least one free occurrence of x in c in a sub-command of the form $x{:=}e$.

Definition 4.4 The set $\mathtt{writes}(c)$ of identifiers occurring freely as targets of assignment in c is given by:

$$\texttt{writes}(\mathbf{skip}) = \emptyset$$

$$\texttt{writes}(x{:=}e) = \{x\}$$

$$\texttt{writes}(c_1; c_2) = \texttt{writes}(c_1 \| c_2) = \texttt{writes}(c_1) \cup \texttt{writes}(c_2)$$

$$\texttt{writes}(\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2) = \texttt{writes}(c_1) \cup \texttt{writes}(c_2)$$

$$\texttt{writes}(\mathbf{while}\ b\ \mathbf{do}\ c) = \texttt{writes}(c)$$

$$\texttt{writes}(\mathbf{await}\ b\ \mathbf{then}\ a) = \texttt{writes}(a)$$

$$\texttt{writes}(\mathbf{local}\ d\ \mathbf{in}\ c) = \texttt{writes}(c) - \texttt{dec}(d)$$

5 Concurrent reads and writes of a context

Next we define, for each parallel context C, the pair $\texttt{crw}(C) = (R, W)$, where R is the set of identifiers which occur free in evaluation contexts concurrent to a hole of C, and W is the set of identifiers occurring free in assigning contexts concurrent to a hole. As usual the definition is inductive. It suffices to work with sets here rather than multisets, since what matters for our present purposes is whether or not the context may change an identifier's value concurrently while whatever command occupies the hole is running, not how many times the context may do this; even once is bad enough.

Definition 5.1 The concurrent-reads-and-writes of a context C are given by:

$$\texttt{crw}([-]) = \texttt{crw}(\mathbf{skip}) = \texttt{crw}(x{:=}e) = (\emptyset, \emptyset)$$

$$\texttt{crw}(C; c_2) = \texttt{crw}(c_1; C) = \texttt{crw}(C)$$

$$\texttt{crw}(\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ C) = \texttt{crw}(\mathbf{if}\ b\ \mathbf{then}\ C\ \mathbf{else}\ c_2) = \texttt{crw}(C)$$

$$\texttt{crw}(\mathbf{while}\ b\ \mathbf{do}\ C) = \texttt{crw}(C)$$

$$\texttt{crw}(\mathbf{await}\ b\ \mathbf{then}\ a) = (\emptyset, \emptyset)$$

$$\texttt{crw}(\mathbf{local}\ d\ \mathbf{in}\ C) = \texttt{crw}(C)$$

$$\texttt{crw}(c \| C) = \texttt{crw}(C \| c) = (R \cup \texttt{reads}(c), W \cup \texttt{writes}(c)),$$
$$\text{where } (R, W) = \texttt{crw}(C)$$

Note that the clause for **local** d **in** C may include in the concurrent reads and writes some of the identifiers declared by d; when code is inserted into the context occurrences of these identifiers become bound, but we still need to know if and how the code uses these identifiers concurrently.

6 Transfer principles

We now state some fundamental properties of trace semantics, which formalize the sense in which the behavior of a parallel program depends only on the values of its free identifiers. We say that two states s and s' *agree* on a set X of identifiers if they map each identifier in this set to (variables which have) the same integer value. These properties are analogues in the parallel setting of “Agreement” properties familiar from the sequential setting. Their proofs are straightforward structural inductions based on the trace semantic definitions.

Theorem 6.1 *Let α be a trace of c and (s, s') be a step of α. Then s agrees with s' on all identifiers not in* $\texttt{writes}(c)$. □

Theorem 6.2 *Let $(s_0, s'_0)(s_1, s'_1)\dots(s_n, s'_n)\dots$ be a trace of c. Then for every sequence of states $t_0, t_1, \dots, t_n, \dots$ such that for all $i \geq 0$, t_i agrees with s_i on $X \supseteq \texttt{reads}(c)$, there is a trace*

$$(t_0, t'_0)(t_1, t'_1)\dots(t_n, t'_n)\dots$$

of c such that for all $i \geq 0$, t'_i agrees with t_i on $X \cup \texttt{writes}(c)$. □

Having set up the relevant background definitions and this key agreement lemma we can now present the transfer principles to which we have been leading.

6.1 A transfer principle for atomic contexts

The first one is almost too obvious to include: it suffices to use sequential reasoning about any code used in a syntactically atomic context. This holds in both coarse- and fine-grained semantics, so we will use $\equiv_{\mathcal{T}}$ to stand for either form of trace equivalence.

Theorem 6.3 *If A is an atomic context and $a_1 \equiv_{\mathcal{M}} a_2$, then $A[a_1] \equiv_{\mathcal{T}} A[a_2]$.*

Proof. The traces of **await** b **then** a depend only on the “atomic” traces of a, i.e. on the traces of a which represent uninterrupted complete executions; and (s, s') is an atomic trace of a iff $(s, s') \in \mathcal{M}(a)$. □

6.2 A sequential transfer principle

The next transfer principle identifies conditions under which sequential equivalence of code fragments can safely be relied upon to establish trace equivalence of parallel programs.

Theorem 6.4 *If $\texttt{free}(c_1) \cup \texttt{free}(c_2) \subseteq \texttt{bound}(C)$, and $(R, W) = \texttt{crw}(C)$, and*

$$|\texttt{reads}(c_i) \cap W| + |\texttt{writes}(c_i) \cap R| = 0, \;\; i = 1, 2$$

then

$$c_1 \equiv_{\mathcal{M}} c_2 \Rightarrow C[c_1] \equiv_{\mathcal{T}} C[c_2].$$ □

It is worth noting that the provisos built into this theorem are essential. If we omit the local declaration around the context the result becomes invalid, since the assumption that c_1 and c_2 are sequentially equivalent is not strong enough to imply that c_1 and c_2 are trace equivalent. And if we try to use the code fragments in a context with which it interacts non-trivially again the result fails: when c_1 and c_2 are sequentially equivalent it does not follow that **local** d **in** $(c \| c_1)$ and **local** d **in** $(c \| c_2)$ are trace equivalent for all c, even if d declares all of the free identifiers of c_1 and c_2. A specific counterexample is obtained by considering the commands

$$c_1 : \quad x{:=}x+1;\ x{:=}x-1$$

$$c_2 : \quad x{:=}x$$

We have $\mathtt{reads}(c_i) = \{\!|x|\!\}$, $\mathtt{writes}(c_i) = \{x\}$. Let C be the context

$$\textbf{local}\ x = 0\ \textbf{in}\ (([-] \| x{:=}2);\ y{:=}x).$$

Then $\mathtt{bound}(C) = \{x\}$ and $\mathtt{crw}(C) = (\emptyset, \{x\})$. Using the notation of the theorem, we have

$$|\mathtt{reads}(c_i) \cap W| = 1, \quad |\mathtt{writes}(c_i) \cap R| = 0$$

so that the assumption is violated. And it is easy to see that $c_1 \equiv_{\mathcal{M}} c_2$, but

$$C[c_1] \equiv_{\mathcal{T}} y{:=}0\,\mathbf{or}\,y{:=}1\,\mathbf{or}\,y{:=}2$$

$$C[c_2] \equiv_{\mathcal{T}} y{:=}0\,\mathbf{or}\,y{:=}2$$

so that $C[c_1] \not\equiv_{\mathcal{T}} C[c_2]$.[3]

Another example shows that the other half of the assumption cannot be relaxed. Consider

$$c_1 : \quad x{:=}1;\ \textbf{while true do skip}$$

$$c_2 : \quad x{:=}2;\ \textbf{while true do skip}$$

Let C be the context

$$\textbf{local}\ x = 0\ \textbf{in}\ ([-] \| y{:=}x).$$

Then $\mathtt{bound}(C) = \{x\}$, $\mathtt{free}(c_i) = \mathtt{writes}(c_i) = \{x\}$, and $\mathtt{reads}(c_i) = \{\!|\,|\!\}$. Moreover $c_1 \equiv_{\mathcal{M}} c_2$, since $\mathcal{M}(c_i) = \{(s, \perp) \mid s \in \mathbf{S}\}$ $(i = 1, 2)$. We have

$$|\mathtt{reads}(c_i) \cap W| = 0, \quad |\mathtt{writes}(c_i) \cap R| = 1$$

so that the assumption is violated again. And we also have

$$C[c_1] \equiv_{\mathcal{T}} (y{:=}0\,\mathbf{or}\,y{:=}1);\textbf{while true do skip}$$

$$C[c_2] \equiv_{\mathcal{T}} (y{:=}0\,\mathbf{or}\,y{:=}2);\textbf{while true do skip}$$

[3] Although our programming language did not include a non-deterministic choice operator $c_1\,\mathbf{or}\,c_2$ it is convenient to use it as here, to specify a command that behaves like c_1 or like c_2; in terms of trace sets we have $\mathcal{T}(c_1\,\mathbf{or}\,c_2) = \mathcal{T}(c_1) \cup \mathcal{T}(c_2)$, a similar equation holiding in coarse- and in fine-grained versions.

so that $C[c_1] \not\equiv_{\mathcal{T}} C[c_2]$.

The above theorem is always applicable in the special case where the context is sequential. We therefore state the following:

Corollary 6.5 *If S is a sequential context, and* $\texttt{free}(c_1) \cup \texttt{free}(c_2) \subseteq \texttt{bound}(S)$, *then*

$$c_1 \equiv_{\mathcal{M}} c_2 \Rightarrow S[c_1] \equiv_{\mathcal{T}} S[c_2].$$

Proof. When S is sequential we can show, by induction on the structure of S, that $\texttt{crw}(S) = (\emptyset, \emptyset)$. □

To illustrate the benefits of these results, note that many simple laws of sequential equivalence are well known, and tend to be taken for granted when reasoning about sequential programs. Note in particular the following instances of de Bakker's laws of (sequential) equivalence [6], which can be used to simplify sequences of assignments:

$$\begin{aligned} x{:=}x &\equiv_{\mathcal{M}} \textbf{skip} \\ x{:=}e_1; x{:=}e_2 &\equiv_{\mathcal{M}} x{:=}[e_1/x]e_2 \\ x_1{:=}e_1; x_2{:=}e_2 &\equiv_{\mathcal{M}} x_2{:=}e_2; x_1{:=}e_1, \\ &\qquad \text{if } x_1 \notin \texttt{free}(e_2)\ \&\ x_2 \notin \texttt{free}(e_1)\ \&\ x_1 \neq x_2 \end{aligned}$$

These laws fail to hold in the parallel setting, and become unsound when $\equiv_{\mathcal{M}}$ is replaced by $\equiv_{\mathit{fine}}$ or $\equiv_{\mathit{coarse}}$. Our result shows the extent to which such laws may safely be used when reasoning about the safety and liveness properties of *parallel* programs, pointing out sufficient conditions under which sequential analysis of key code fragments is enough to ensure correctness of a parallel program.

6.3 A coarse- to fine-grained transfer principle

Finally, we now consider what requirements must be satisfied in order to safely employ coarse-grained trace-based reasoning in establishing fine-grained equivalences. This may be beneficial, as remarked earlier, since for a given code fragment the coarse-grained trace set forms a (usually proper) subset of the fine-grained trace set and may therefore permit a streamlined analysis. This is especially important for code which may be executed concurrently, since it may help minimize the combinatorial analysis. Indeed, Andrews [1] supplies a series of examples of protocols in which a "fine-grained" solution to a parallel programming problem (such as mutual exclusion) is derived by syntactic transformation from a "coarse-grained" solution whose correctness is viewed as easier to establish. Common to all of these examples is the desire to appeal to coarse-grained reasoning when trying to establish correctness in the fine-grained setting.

We begin with a so-called "at-most-once" property that Andrews uses informally to facilitate the analysis and development of a collection of mutual exclusion protocol designs. The relevant definitions from Andrews, adapted to our setting, are as follows:

- An expression b (or e) has the at-most-once property if it refers to at most one identifier that might be changed by another process while the expression is being evaluated, and it refers to this identifier at most once.
- An assignment $x{:=}e$ has the at-most-once property if either e has the at-most-once property and x is not read by another process, or if e does not refer to any identifier that may be changed by another process.
- A command c has the at-most-once property if every assignment and boolean test occurring non-atomically in c has the at-most-once property.

An occurrence is atomic if it is inside a subcommand of form **await** b **then** a.

Andrews's methodology is based on the idea that if a command has the at-most-once property then it suffices to assume coarse-grained execution when reasoning about its behavior, since there will be no discernible difference with fine-grained execution. However, the above characterization of an at-most-once property is only informal and slightly imprecise, in particular in relying on implicit analysis of the context in which code is to be executed. We will couch our transfer principle in slightly more specific but general terms based on a precise reformulation of this property, referring to the `crw` definition from above.

Theorem 6.6 *If* $\mathtt{free}(c_1) \cup \mathtt{free}(c_2) \subseteq \mathtt{bound}(C)$, *and* $(R, W) = \mathtt{crw}(C)$, *and*

$$\begin{array}{rl} \textit{either} & |\mathtt{reads}(c_i) \cap W| = 0 \\ \textit{or} & |\mathtt{reads}(c_i) \cap W| = 1 \ \&\ |\mathtt{writes}(c_i) \cap (R \cup W)| = 0,\ i = 1, 2 \end{array}$$

then

$$c_1 \equiv_{coarse} c_2 \Rightarrow C[c_1] \equiv_{fine} C[c_2]. \qquad \square$$

Thus our formal version of the at-most-once property can be read as requiring that the command reads at most one occurrence of an identifier written concurrently by the context, and if it reads one then none of its writes affect any identifier which is either read or written concurrently by the context. Our insistence in the above theorem that the code being analyzed (c_1 and c_2) only affects local variables, i.e. identifiers which become bound when the code is inserted into the context, is reflected in Andrews's setting by an assumption that all processes have local registers.

Again we show that the built-in provisos imposing locality and the at-most-once property cannot be dropped.

Firstly, every program has the at-most-once property, trivially, for the context $[-]$. But the assumption that $c_1 \equiv_{coarse} c_2$ is insufficient to ensure that $c_1 \equiv_{fine} c_2$. Thus the result becomes invalid if we omit the localization around the context.

To illustrate the need for the at-most-once assumption, let the programs c_1 and c_2 be $y{:=}x + x$ and $y{:=}2 \times x$. These programs are clearly coarsely equivalent. Let C be the context

$$\textbf{local}\ x = 0; y = 0\ \textbf{in}\ (([-]\|x{:=}1);\ z{:=}y).$$

Of course c_1 refers twice to x, which is assigned to by the context concurrently; c_1 does not satisfy the at-most-once property for C. Moreover we can see that

$$\textbf{local}\ x = 0; y = 0\ \textbf{in}\ ((y{:=}x + x\|x{:=}1); z{:=}y) \equiv_{\mathit{fine}} z{:=}0\,\textbf{or}\,z{:=}1\,\textbf{or}\,z{:=}2$$

$$\textbf{local}\ x = 0; y = 0\ \textbf{in}\ ((y{:=}2 \times x\|x{:=}1); z{:=}y) \equiv_{\mathit{fine}} z{:=}0\,\textbf{or}\,z{:=}2$$

so that $C[c_1] \not\equiv_{\mathit{fine}} C[c_2]$.

Also note that the other way for the assumption to fail is when c_1 (say) both reads and writes to a concurrently accessed identifier. For instance, let c_1 be $x{:=}x$ and c_2 be **await true then** $x{:=}x$. Let C be the context

$$\textbf{local}\ x = 0\ \textbf{in}\ (([-]\|x{:=}1); y{:=}x)$$

Then we have $|\texttt{reads}(c_i) \cap W| = 1$ and $|\texttt{writes}(c_i) \cap (R \cup W)| > 0$. And $c_1 \equiv_{\mathit{coarse}} c_2$. But $C[c_1] \equiv_{\mathit{fine}} y{:=}0\,\textbf{or}\,y{:=}1$, and $C[c_2] \equiv_{\mathit{fine}} y{:=}1$.

It is also worth remarking that the above principle cannot be strengthened by replacing the assumption that c_1 and c_2 are *coarsely equivalent* with the weaker assumption that c_1 and c_2 are *sequentially equivalent*. For example, let c_1 and c_2 be

$$y{:=}1;\ \textbf{while true do skip}$$

and

$$y{:=}2;\ \textbf{while true do skip}.$$

Let C be the context **local** $y = 0$ **in** $([-]\|z{:=}y)$. Then we have $\texttt{reads}(c_i) = \emptyset$, $\texttt{writes}(c_i) = \{y\}$, $\texttt{crw}(C) = (\{y\}, \{z\})$, $\texttt{bound}(C) = \{y\}$. Moreover, $c_1 \equiv_{\mathcal{M}} c_2$ holds, since $\mathcal{M}(c_i) = \{(s, \perp) \mid s \in \mathbf{S}\}$, $i = 1, 2$. However,

$$C[c_1] \equiv_{\mathit{fine}} (z{:=}0\ \textbf{or}\ z{:=}1)$$

and

$$C[c_2] \equiv_{\mathit{fine}} (z{:=}0\ \textbf{or}\ z{:=}2),$$

so that $C[c_1] \not\equiv_{\mathit{fine}} C[c_2]$.

The coarse- to fine-grained transfer theorem given above generalizes some more *ad hoc* arguments based on occurrence-counting in Andrews's book, resulting in a single general principle in which the crucial underlying provisos are made explicit. To make the connection with Andrews's examples more precise, note the following special cases of our theorem, which appear in paraphrase in Andrews:

- If b refers at most once to identifiers written concurrently (by the context), then **await** b **then skip** can be replaced by **while** $\neg b$ **do skip** (throughout the program). This rule may be used to justify replacement of a conditional atomic action with a (non-atomic) busy-wait loop.

- If $x{:=}e$ has the at-most-once property (for the context) then the assignment $x{:=}e$ can be replaced by its atomic version **await true then** $x{:=}e$ (throughout the program). This rule may be used to simplify reasoning about the potential for interaction between processes.

7 Conclusions and future work

We have identified conditions under which it is safe to employ "sequential" reasoning about code fragments while trying to establish "parallel" correctness properties such as safety and liveness. We have also identified conditions governing the safe use of coarse-grained reasoning in proving fine-grained properties.

These transfer principles can be seen as supplying a semantic foundation for some of the ideas behind Andrews's protocol analysis, and a potential basis for further generalization and the systematic development of techniques to permit easier design and analysis of parallel programs. We plan to extend our ideas and results to cover a wider variety of examples, including some of the protocols discussed by Dingel. It would also be interesting to explore the relationship between our approach and Dingel's notion of context-sensitive approximation.

These results permit the use of a simpler, more abstract semantics, together with a notion of semantic equivalence which is easier to establish, to facilitate reasoning about the behavior of a parallel system. It would be interesting to investigate the possible utility of transfer principles in improving the efficiency of model-checking for finite-state concurrent systems.

8 Acknowledgements

The anonymous referees made a number of helpful suggestions. The author would also like to thank his former Ph.D. student, Jürgen Dingel, whose thesis research provides a stimulus for the work reported here.

References

[1] Andrews, G., "Concurrent Programming: Principles and Practice," Benjamin/Cummings (1991).

[2] Brookes, S., *Full abstraction for a shared-variable parallel language,* Information and Computation, **127**(2), 145–163 (June 1996).

[3] Brookes, S., *The essence of Parallel Algol.* Proc. 11th IEEE Symposium on Logic in Computer Science, IEEE Computer Society Press, 164–173 (1996). To appear in *Information and Computation.*

[4] Brookes, S., *Idealized CSP: Combining Procedures with Communicating Processes.* Mathematical Foundations of Programming Semantics, 13^{th} Conference, March 1997, Electronic Notes in Theoretical Computer Science **6**, Elsevier Science (1997).
URL: `http://www.elsevier.nl/locate/entcs/volume6.html`.

[5] Brookes, S., *Communicating Parallel Processes.* In: *Millenium Perspectives in Computer Science*, Proceedings of the Oxford-Microsoft Symposium in honour of Professor Sir Antony Hoare, edited by Jim Davies, Bill Roscoe, and Jim Woodcock, Palgrave Publishers (2000).

[6] de Bakker, J., *Axiom systems for simple assignment statements.* In *Symposium on Semantics of Algorithmic Languages*, edited by E. Engeler. Springer-Verlag LNCS **181**, 1–22 (1971).

[7] Dingel, J., "Systematic Parallel Programming," Ph. D. thesis, Carnegie Mellon University, Department of Computer Science (May 2000).

[8] Jones, C. B., *Tentative steps towards a development method for interfering programs*, ACM Transactions on Programming Languages and Systems **5**(4):576–619 (1983).

[9] Park, D., *On the semantics of fair parallelism.* In "Abstract Software Specifications," edited by D. Bjørner, Springer-Verlag LNCS **86**, 504–526 (1979).

[10] Park, D., "Concurrency and Automata on Infinite Sequences," Springer LNCS **104** (1981).