Derivation of Concurrent Algorithms in Tempo

Steve Gregory

Department of Computer Science
University of Bristol
Bristol BS8 1TR, England

steve@cs.bris.ac.uk

Abstract

Tempo is a logic programming language that has recently been designed with the aim of allowing concurrent programs to be written in a more declarative manner than existing languages. One of the benefits of declarative programming is the potential for systematic derivation of programs — the subject of this paper. Here we present a few transformation rules that can be applied to Tempo programs, and then describe in detail the derivation of some concurrent algorithms in Tempo. An outline of the Tempo language is also included in order to make the paper self-contained.

1 Introduction

One of the major advantages of the declarative approach to programming is the ability to construct programs systematically from specifications by applying correctness-preserving transformation rules. Existing programs can be transformed to new, possibly more efficient, programs in the same way. Although it could be argued that the same technique can also be applied to "conventional" languages, it is undoubtedly far easier in a declarative context. For example, a logic program (in some executable logic programming language) can be derived from a logic specification by applying the standard rules of logic that have been established for thousands of years.

The feasibility of program transformation, along with the many other attractions of declarative programming, depends upon the target programming language being both executable and declarative. Unfortunately, most logic programming languages are only partially declarative, in that not all relevant aspects of a program's behaviour are explicit in the program. One example is termination, which, even in "pure" Prolog, is dependent on control considerations such as the order of goals and clauses: if one program P terminates, another, logically equivalent, program P' may not. The situation is made much worse by the use of non-logical primitives such as I/O, which are indispensable in "real" programs.

In this paper we address concurrent programming, a subject of increasing importance with the advent of multiprocessor architectures and computer networks. Concurrent programs comprise several processes which interact (synchronize and communicate) with each other. Programs that interact with their environment, such as the processes in a concurrent program, are often termed *reactive*. Unlike a *transformational* program, which merely terminates with a final result, a reactive program produces results incrementally throughout its execution and may not even be intended to terminate. A transformational program is considered correct if its final result is correct (partial correctness) and the program terminates. In contrast, a reactive program may be required to satisfy several properties *during its execution*.

These are usually classified into *safety* properties, those that must *always* be true, and *progress* (or *liveness*) properties, those that must *eventually* hold. Partial correctness and termination are special cases of safety and progress properties, respectively.

Concurrent logic programming languages such as Parlog (Gregory, 1987) and KL1 (Ueda and Chikayama, 1990) are declarative in the same sense as Prolog: a program explicitly describes its final result. Unfortunately, all other important (safety and progress) properties of a program are implicit and have to be preserved by proper use of control features (modes, sequencing, etc.). It follows that, if a concurrent logic program is transformed to a logically equivalent one, the only property that is guaranteed to be preserved is the final result that the program computes. This is not a very significant property in reactive programs, especially those that never terminate! Perhaps because operational semantics play such a prominent role in concurrent logic programs, there has been a trend in recent years to abandon the goal of "declarativeness" altogether. A few languages have evolved, e.g., (Foster *et al*., 1992; McCabe and Clark, 1995), which incorporate various aspects of Parlog in an imperative framework.

In our view, it is premature to abandon the ideal of declarative concurrent programming: instead of removing the "logic" from concurrent logic programming, we should attempt to design languages that are *more* declarative than existing ones. We have moved a step in this direction with the design of Tempo (Gregory and Ramirez, 1995). Tempo is a concurrent programming language based on classical first-order logic which allows *all safety properties* to be stated declaratively, not only the program's final result.

In the next section we give a brief outline of Tempo, a full description of which can be found in (Gregory and Ramirez, 1995). Section 3 demonstrates how a specification of a classic concurrent programming problem can be transformed into various algorithms, all guaranteed to have the specified safety properties. The original contribution of this paper is the transformation method described and illustrated in Section 3, the results of which were stated without justification in (Gregory and Ramirez, 1995). The detail presented here should be sufficient to show how algorithms for other problems can be derived in Tempo.

2 Outline of Tempo

In Tempo, as in most logic programming languages, a program is a set of Horn clauses. The arguments of the head of each clause are distinct variables, so we write a *constraint definition* (a set of clauses defining a predicate) as a single clause containing a disjunction:

$$H \leftarrow Cs_1 \texttt{;} \ \ldots \texttt{;} \ Cs_n.$$

(The disjunction operator '`;`' is less tightly binding than '`,`' but more tightly binding than '←'.) Each constraint definition is a complete definition of the relation being defined, so the above is equivalent to

$$H \leftrightarrow \exists V \ Cs_1 \vee \ldots \vee Cs_n$$

where V denotes the variables appearing in Cs_1, ..., Cs_n but not in H. A further syntactic restriction is that no constants or function symbols are allowed in clauses.

Precedence Constraints

The constraints in a Tempo program describe relations between *events*. Events are atomic (i.e., they have no duration), so the main attribute of an event is its *execution time*. The only primitive predicate in Tempo is the precedence constraint $X < Y$, which is read "X precedes Y" and indicates that event X occurs chronologically before event Y. Mutually unconstrained events may occur in any order, or even simultaneously. The absolute execution time of an event cannot be specified, with one exception: if $X < X$, the execution time of X must be the special value *eternity*; in this case, we call X an *eternity event*. Eternity events are *never* executed.

Figure 1 defines the '<' predicate more formally ((4) is a corollary of (1–3)): '<' is transitive and no event precedes itself except an eternity event, which is preceded by *every* event.

$$\forall X \forall Y \forall Z \ (X < Y \wedge Y < Z \ \rightarrow \ X < Z) \tag{1}$$
$$\forall X \forall Y \ (\mathit{time}(Y,\mathit{eternity}) \ \rightarrow \ X < Y) \tag{2}$$
$$\forall X \ (X < X \ \rightarrow \ \mathit{time}(X,\mathit{eternity})) \tag{3}$$
$$\forall X \forall Y \ (\mathit{time}(Y,\mathit{eternity}) \wedge Y < X \ \rightarrow \ \mathit{time}(X,\mathit{eternity})) \tag{4}$$

Figure 1: Properties of the '<' predicate

A Tempo program can describe processes (or *activities*) as partially ordered sets of atomic *events*. Events often represent transitions between successive states. For example, query (Q1) represents a *thinker* activity, which is initially in thinking state, becomes hungry (event `H`), starts eating (event `E`), and finally resumes thinking forever (event `T`):

```
←  thinker(H,E,T).                                    (Q1)

thinker(H,E,T) ← H < E, E < T.
```

The solution to query (Q1) is any totally ordered set of events such that `H < E < T`. Notice that some of these could be eternity events. Indeed, an answer substitution in which *all* events are eternity events is a valid solution to *any* Tempo query.

Operational Semantics

Although Tempo uses the familiar syntax of Horn clauses, its operational semantics is completely different from that of conventional logic programming languages such as Prolog or Parlog. Instead of merely returning an answer substitution for query variables, a Tempo interpreter actually *executes* the specified activities:

1. Instead of computing an execution time `t` for event `E`, Tempo executes an event (which we shall refer to by the variable name `E`) at time `t`. This way, events are executed in the specified order, but their execution times are implicit. For example, query (Q1) results in a sequence "*execute* `H`; *execute* `E`; *execute* `T`", in that order. An event that has an execution time of *eternity* is never executed; this corresponds to termination or deadlock.

2. The solution is constructed incrementally in ascending time order. That is, an event is executed as soon as its predecessors have been executed, without waiting for a complete solution. This yields useful results even if the set of events is large or infinite.

A Tempo interpreter works essentially as follows. Each event `E` in a query is checked to see whether it is preceded by another event, i.e., if a constraint `X < E` exists. If `E` is not preceded by any other event (it is *enabled*), it can be executed, whereupon all constraints of the form `E < F` are deleted and each event `F` (previously preceded by `E`) is checked in the same way to see whether it is enabled.

User-Defined Constraints

While checking whether an event `E` is enabled, any user-defined constraint that has `E` as an argument (such as `thinker(H,E,T)`) is *expanded*, i.e., replaced by its body; this is repeated until `E` appears only in primitive '<' constraints.

Event Trees

Many activities comprise a large or infinite set of events, for example a *philosopher*, which repeatedly cycles through the three states thinking, hungry, and eating. We cannot name each event individually in a finite specification, so instead we define relations on *trees* of events. Each event `E` may have one or more *offspring* events associated with it, which, for operational reasons, must be preceded by `E` itself. The offsprings of `E` are named `E+1`, `E+2`, etc., and may appear in queries and in bodies (but not heads) of constraint definitions. For example, the following recursive definition of `phil` represents a philosopher activity:

```
phil(H,E,T) ← H <* E, E <* T, T <* H+1.

X <* Z ← X < Z, X+1 <* Z+1.
```

The '+' notation can be defined formally by the following axioms:

$$p(s_1,\ldots,s_i,X{+}N,t_1,\ldots,t_j) \leftrightarrow (\ \exists Y\ \mathit{offs}(X,N,Y) \wedge X < Y \wedge p(s_1,\ldots,s_i,Y,\ t_1,\ldots,t_j)\ )$$

$$Y = X{+}N\ \wedge\ Z = X{+}N\ \rightarrow\ Y = Z$$

where $\mathit{offs}(X,N,Y)$ indicates that event Y is the Nth offspring of event X.

Disjunction

Disjunction is necessary to describe some problems. For example, in the mutual exclusion example that we describe below, it is necessary to ensure that one of two philosophers starts thinking before the other begins to eat. The following `mutex` constraint uses a disjunction to express this requirement:

```
mutex(E1,E2,T1,T2) ←  T1 < E2, mutex(E1+1,E2,T1+1,T2);
                      T2 < E1, mutex(E1,E2+1,T1,T2+1).
```

A Tempo program with disjunction naturally defines a search tree in which each branch admits a different ordering of events: each execution of the program must follow a single branch of this tree. Rather than attempting to perform an intelligent planning-like search for a suitable ordering, Tempo uses a special "shallow search" strategy to select a branch, which works as follows.

If a disjunctive constraint is encountered while checking whether an event `E` is enabled, each alternative of the constraint is checked. If `E` is enabled in all of the disjunction's alternatives, `E` is enabled by the disjunction. If `E` is not enabled in any alternative, `E` is not enabled. However, if `E` is enabled in some but not all of the alternatives, the disjunction is *reduced*: the alternatives that do not enable `E` are deleted, and E is enabled by the disjunction. This step is performed only if all other, non-disjunctive, constraints have also enabled E, and therefore E can be executed.

Concurrent Execution

A Tempo query can be solved cooperatively by a number of workers, or *processes*. A query may take the general form

$$\leftarrow \ \{Cs_1\},\ \ldots,\ \{Cs_m\}.$$

where each Cs_i is a conjunction of constraints which is executed as a separate process. This is logically identical to the same query without the braces: it will execute the specified events in an order that satisfies the constraints in the query. The difference is purely operational: each process solves the constraints allocated to it, independently of other processes except where the processes share an event. A shared event is executed only after it has been enabled by *every* process that shares it. Shared events provide the sole mechanism for Tempo processes to communicate.

Values

The language that we have described above, and in (Gregory and Ramirez, 1995), has no explicit representation of data. However, the latest version of Tempo includes *values* (numbers, atoms, and structured terms), which can appear as arguments of constraints and can also be associated with events. This allows computation, as well as synchronization, to be expressed in Tempo. Some points mentioned above (e.g., that '<' is the only primitive) do not apply to the extended language. The extended language features will be fully described in a forthcoming paper; we shall not need to use them in the remainder of this paper.

An Example: Mutual Exclusion

The mutual exclusion problem is a classic concurrent programming example. Given a number of (philosopher) processes repeatedly executing two alternate sequences of instructions — a non-critical section (*thinking*) and a critical section (*eating*) — ensure that no more than one process executes its critical section at any time. This is a safety property. There are also various progress properties that a solution to this problem should satisfy: for example, that a philosopher that is trying to enter its critical section should eventually be able to do so.

The following Tempo query (Q2) succinctly expresses the mutual exclusion problem for two philosophers:

```
←  phil(H1,E1,T1), phil(H2,E2,T2), mutex(E1,E2,T1,T2).        (Q2)
```

This is not only a straightforward "specification" of the problem, but also a Tempo program. When the program is run, the events corresponding to H1, E1, T1, etc. are executed in any order that satisfies the specification: in particular, the intervals [E1..T1], [E1+1..T1+1], ..., [E2..T2], [E2+1..T2+1], etc. do not overlap with each other. In a practical application, each event might be associated with a useful action: for example, E1 and T1 could mark the beginning and end of an operation that accesses some shared physical resource.

Because the safety properties (i.e., the order of events) are explicit in the program, they are guaranteed to be preserved by any Tempo implementation. Likewise, the order of constraints in a conjunction, and alternatives in a disjunction, is irrelevant. However, progress properties (in this example, the requirement that any process trying to enter its critical section should eventually succeed) are *not* expressed explicitly: as in most languages, these have to be preserved by reasoning about the operational behaviour of Tempo.

3 Deriving Algorithms in Tempo

The definition *Algorithm* = *Logic* + *Control* (Kowalski, 1979) applies to any logic programming language. In the case of Tempo, we argue that the *Logic* component of a program captures more aspects of behaviour than in most languages, but the *Control* component is still significant. That is, the algorithm that a Tempo program realizes depends not only on the constraints that it contains but also on the control "annotations" used. In Prolog (but not in Tempo), even changing the order of goals or clauses can result in a different algorithm, though (in both languages) more substantial changes are usually required to transform one algorithm to another.

A Tempo programmer can write concurrent algorithms by using the `{}` syntax outlined above. For the mutual exclusion example, query (Q2) is actually a sequential algorithm that simulates a solution. To turn it into a concurrent algorithm, the `phil` constraints, each representing a philosopher, need to be allocated to different processes. A simple way to do this is by annotating query (Q2) as below:

```
← {phil(H1,E1,T1)}, {phil(H2,E2,T2)}, {mutex(E1,E2,T1,T2)}.
```
(Q3)

Query (Q3) represents an algorithm in which the first two (`phil`) processes each communicate with a third (`mutex`) process. When the first philosopher wants to "eat" it enables event `E1`. `E1` can only be executed when it has also been enabled by the `mutex` process, effectively giving the first philosopher permission to eat. When this happens, the `mutex` process commits to one of the two alternatives of its disjunction: because this includes the constraint `T1 < E2`, the second philosopher will not be able to start eating until the first one finishes (and enables `T1`).

Concurrent programs are often intended to be executed in parallel, with the constituent processes possibly running on distinct physical processors. One of the most challenging aspects of concurrent programming is the wide variety of parallel architectures that exist, and the corresponding proliferation of concurrent programming paradigms and languages. For most problems, there is no universal concurrent algorithm that is well suited to all types of parallel architecture. For example, an algorithm in which processes communicate by shared variables may be efficient when run on a shared memory multiprocessor but not on a distributed machine. Also, an algorithm that involves a lot of interprocess communication may well be inefficient on a distributed architecture, where communication is expensive.

For example, query (Q3) is a perfectly correct concurrent Tempo program for mutual exclusion, but it is not necessarily efficient because:

1. The `mutex` process, which communicates with all `phil` processes, may constitute a bottleneck, especially if scaled up to a large number of philosophers.

2. The communication between a `phil` process and the `mutex` process is analogous to synchronous message passing: a shared event is executed when both processes enable it. Synchronous message passing is directly provided in some multiprocessors, but may be expensive to implement in others.

The algorithm that query (Q3) realizes is shown in an imperative style in Figure 2(a), which is reproduced from (Gregory and Ramirez, 1995). The processes named `c1`, `c2`, and `s` in Figure 2(a) correspond to the first, second, and third groups of constraints in (Q3), respectively.

(a)

```
% {phil(H1,E1,T1)}
process c1:
repeat
    think;
    (H1)
    send eat to s (E1);
    eat;
    send think to s (T1);
forever
```

```
% {mutex(E1,E2,T1,T2)}
process s:
repeat
    receive eat from c1 (E1) →
       receive think from c1 (T1)
or  receive eat from c2 (E2) →
       receive think from c2 (T2)
forever
```

(b)

```
% {phil(H1,Eb1,T1),
   E1 <* Eb1}
process c1:
repeat
    think;
    send hungry to s (H1);
    receive eat from s (E1);
    eat (Eb1);
    send think to s (T1);
forever
```

```
% {H1 <* E1, H2 <* E2,
   mutex(E1,E2,T1,T2)}
process s:
repeat
    receive hungry from c1 (H1) →
       send eat to c1 (E1);
       receive think from c1 (T1)
or  receive hungry from c2 (H2) →
       send eat to c2 (E2);
       receive think from c2 (T2)
forever
```

(c)

```
Initially: b1 := false; b2 := false;
```

```
% {phil(H1,E1,T1)}
process c1:
repeat
    think;
    b1 := true (H1);
    repeat until not b2 (E1);
    eat;
    b1 := false (T1);
forever
```

```
% {phil(H2,E2,T2)}
process c2:
repeat
    think;
    b2 := true (H2);
    repeat until not b1 (E2);
    eat;
    b2 := false (T2);
forever
```

Figure 2: Mutual exclusion algorithms — (a) synchronous message passing; (b) asynchronous message passing; (c) shared variables

Below we show how a Tempo program (or specification) can be transformed into another one that may be more suitable for a particular parallel architecture. Given a Tempo query $\leftarrow q$, our method is to derive a stronger query $\leftarrow q'$ (such that q' logically implies q), which will then be guaranteed to preserve all of the safety properties that $\leftarrow q$ satisfies. The objective is to obtain a Tempo program that can be executed more efficiently on the target machine than the original. Alternatively, the final Tempo program can be translated directly into a conventional language that has an efficient implementation on the target machine.

Transformation Rules

Here we outline the transformation rules that we use. These are rather standard, except for the last two, which are explained in more detail.

1. Unfolding

Given a Tempo constraint definition

$$H \leftarrow Cs.$$

any constraint C in a conjunction such that $C = H\theta$ can be replaced by $Cs\theta$. Because $H \leftrightarrow Cs$, the new conjunction is equivalent to the old one.

2. Folding

Given a Tempo constraint definition

$$H \leftarrow Cs.$$

if a conjunction contains an expression Cs' which is an instance of Cs (i.e., $Cs' = Cs\theta$), then Cs' can be replaced by $H\theta$, because $H \leftrightarrow Cs$. Again, the new conjunction is equivalent to the old one.

3. Adding redundant constraints

Any constraint that is implied by the existing constraints in a conjunction can be added to it without changing its meaning.

4. Deleting subsumed alternatives

An alternative can be removed from a disjunction if it implies at least one of the other alternatives, because $(a \vee b \leftrightarrow a)$ if $(b \rightarrow a)$. Therefore, whenever we derive a constraint definition of the form:

$$p(X_1,\ldots,X_m) \leftarrow r_1;\ r_2;\ \ldots;\ r_n;\ s.$$

such that, for some i, $s \rightarrow r_i$, we can simplify it to:

$$p(X_1,\ldots,X_m) \leftarrow r_1;\ r_2;\ \ldots;\ r_n.$$

5. The unfold/fold rule

Our most sophisticated transformation rule is as follows: In order to replace an expression Cs in the query by Cs', we first define two new predicates of the form:

$$p(X_1,\ldots,X_m) \leftarrow Cs. \qquad (1)$$

$$p'(X_1,\ldots,X_m) \leftarrow Cs'. \qquad (2)$$

where $X_1,\ldots,X_m$ are all of the free variables in Cs and Cs'. We then use standard *totally correct* transformation rules (e.g., rules 1 to 4 above) to transform (1) and (2) into a pair of constraint definitions of the form (1') and (2') below:

$$p(X_1,\ldots,X_m) \leftarrow q,\ r;\ s. \qquad (1')$$

$$p'(X_1,\ldots,X_m) \leftarrow q',\ r';\ s'. \qquad (2')$$

where r' and s' are identical to r and s respectively, but with every occurrence of p replaced by p'. Then, if $q' \rightarrow q$:

$p'(X_1,...,X_m) \rightarrow p(X_1,...,X_m)$

It is possible to prove this fact by induction on the number of constraints for predicate p appearing in r and s.[1] (Recall that the Tempo connective '←' in (1, 2, 1', 2') actually denotes equivalence '↔'.)

It follows that $Cs' \rightarrow Cs$, and therefore we can replace Cs in the query by Cs', to produce a new query that logically implies the old one.

This rule is very similar to the unfold/fold method for proving the validity of replacement laws in the transformation of logic programs[2] (Boulanger and Bruynooghe, 1993; Proietti and Pettorossi, 1994). The difference is that Proietti and Pettorossi derive clauses for (1') and (2') which are *identical*, modulo predicate renaming, to prove (roughly) that $Cs' \leftrightarrow Cs$. We have a weaker requirement, that $q' \rightarrow q$, because we only need to prove that $Cs' \rightarrow Cs$.

An Algorithm for Asynchronous Message Passing

If each shared event appears on the right of a '<' constraint in only one process (the sender) in a Tempo query, the interprocess communication is analogous to asynchronous message passing. Because the receiver cannot delay the event's execution, it is executed as soon as the sender enables it.

Query (Q3) does not satisfy this restriction because `E1` and `E2` each appear on the right of '<' in two processes. However, we can construct a Tempo query (Q4) of the form ← *qa*, such that *qa* logically implies the specification *s*, which satisfies the restrictions corresponding to asynchronous message passing. The specification *s* is repeated below from (Q2):

```
←  phil(H1,E1,T1), phil(H2,E2,T2), mutex(E1,E2,T1,T2).
```

First, the two `phil` constraints are unfolded:

```
←  H1 <* E1, E1 <* T1, T1 <* H1+1,
   H2 <* E2, E2 <* T2, T2 <* H2+1, mutex(E1,E2,T1,T2).
```

Then a new event `Eb1` is introduced between the two events `E1` and `T1`; likewise, `Eb2` is introduced between `E2` and `T2`. This is done by replacing the constraint `E1 <* T1` by `E1 <* Eb1, Eb1 <* T1`, and replacing `E2 <* T2` by `E2 <* Eb2, Eb2 <* T2` (we justify the validity of this replacement below). The query becomes:

```
←  H1 <* E1, E1 <* Eb1, Eb1 <* T1, T1 <* H1+1,
   H2 <* E2, E2 <* Eb2, Eb2 <* T2, T2 <* H2+1,
   mutex(E1,E2,T1,T2).
```

We now add two redundant constraints to the query: `H1 <* Eb1` and `H2 <* Eb2` (these are subsumed by the existing constraints):

```
←  H1 <* E1, E1 <* Eb1, H1 <* Eb1, Eb1 <* T1, T1 <* H1+1,
   H2 <* E2, E2 <* Eb2, H2 <* Eb2, Eb2 <* T2, T2 <* H2+1,
   mutex(E1,E2,T1,T2).
```

Folding, using the definition of the `phil` constraint, yields:

```
←  H1 <* E1, E1 <* Eb1, phil(H1,Eb1,T1),
   H2 <* E2, E2 <* Eb2, phil(H2,Eb2,T2), mutex(E1,E2,T1,T2).
```

---

1 Thanks to John Gallagher for this suggestion.

2 Thanks again to John Gallagher for drawing my attention to this work.

Finally, this conjunction can be arranged into three groups of constraints as in the Tempo query (Q4). Event `E1` now appears on the right of '<' only in the third process: it represents a "permission to eat" message sent by that process to the first philosopher. `Eb1` is an event local to the first philosopher process, representing the act of "beginning to eat". Query (Q4) realizes the algorithm shown in Figure 2(b); the three groups of constraints correspond to processes `c1`, `c2`, and `s`, respectively.

```
←  {phil(H1,Eb1,T1), E1 <* Eb1},
   {phil(H2,Eb2,T2), E2 <* Eb2},
   {H1 <* E1, H2 <* E2, mutex(E1,E2,T1,T2)}.                    (Q4)
```

The correctness of replacing `E1 <* T1` by `(E1 <* Eb1, Eb1 <* T1)` is justified by our unfold/fold rule. We first define a new predicate `pr`:

```
pr(X,Z) ← X <* Y, Y <* Z.
```

Now the two '<*' constraints are unfolded, and the right side is folded using the definition of `pr`, to obtain the recursive constraint definition:

```
pr(X,Z) ← X < Y, Y < Z, pr(X+1,Z+1).
```

while the '<*' constraint is defined as:

```
X <* Z ← X < Z, X+1 <* Z+1.
```

From (1) in Figure 1, we have

```
X < Y, Y < Z → X < Z
```

and so, by applying our unfold/fold rule:

```
pr(X,Z) → X <* Z
```

or, equivalently:

```
X <* Y, Y <* Z → X <* Z
```

An Algorithm for Shared Mutable Variables

Tempo can also simulate communication via shared mutable variables: variables that can be destructively modified and read by several processes. Each variable is represented by a `var` constraint (contained in its own process) and other processes are allowed to share events only with `var` processes: these shared events represent the actions of assigning a value to, and testing the current value of, the mutable variable. For example, a boolean variable, initially false, could be represented by `var(AT,AF,OT,OF)`, where `AT` and `AF` represent assignments to true and false, respectively, and `OT` and `OF` represent observations that the variable has these respective values[3]. Assuming that the assignments to true and false strictly alternate, beginning with an assignment to true, and we only want to observe the false values, the variable can be represented by a simpler constraint `varf(AT,AF,OF)`, defined in Tempo as follows:

```
varf(AT,AF,OF) ← OF < AT, varf(OF+1,AT,AF);
                 AF < OF, varf(OF,AT+1,AF+1).
```

---

3 In the latest version of Tempo, a mutable variable taking an *arbitrary* value can be represented by `var(A,O)`, where `A` and `O` are events with associated values.

The point of this transformation is that any Tempo program that fits this structure can be implemented directly by shared memory hardware: the constraint `varf(AT,AF,OF)` can be enforced by a shared mutable variable `b`, where the events `AT`, `AF`, and `OF` are replaced by the actions "`b:=true`", "`b:=false`", and "`wait until not b`", respectively.

An algorithm for mutual exclusion that uses shared mutable variables is shown in Figure 2(c). This is a simplified version of Peterson's (1981) algorithm with the possibility of deadlock. To render this algorithm in Tempo, we introduce a `varf` constraint for each of the shared variables `b1` and `b2`, as in query (Q5). The first two groups of constraints in (Q5) correspond to processes `c1` and `c2` in Figure 2(c).

```
←  {phil(H1,E1,T1)}, {phil(H2,E2,T2)},
   {varf(H1,T1,E2)}, {varf(H2,T2,E1)}.                    (Q5)
```

It is not at all obvious that the algorithm of Figure 2(c) satisfies the safety properties of mutual exclusion, but this is much easier to prove when the algorithm is expressed in Tempo as in query (Q5). We simply need to transform the specification

```
←  phil(H1,E1,T1), phil(H2,E2,T2), mutex(E1,E2,T1,T2).     (s)
```

into the query

```
←  phil(H1,E1,T1), phil(H2,E2,T2),
   varf(H1,T1,E2), varf(H2,T2,E1).                         (q)
```

We do this in a single step, using the unfold/fold rule: We first define two new predicates `ms` (equivalent to the specification) and `mq` (equivalent to the query):

```
ms(E1,E2,T1,T2,H1,H2) ←
    phil(H1,E1,T1), phil(H2,E2,T2), mutex(E1,E2,T1,T2).

mq(E1,E2,T1,T2,H1,H2) ←
    phil(H1,E1,T1), phil(H2,E2,T2),
    varf(H1,T1,E2), varf(H2,T2,E1).
```

and then use the unfold/fold rule to show that

```
mq(E1,E2,T1,T2,H1,H2) → ms(E1,E2,T1,T2,H1,H2)
```

The first step is to transform the definition of `mq` by unfolding the `varf` constraints:

```
mq(E1,E2,T1,T2,H1,H2) ←
    phil(H1,E1,T1), phil(H2,E2,T2),
    (E2 < H1, varf(H1,T1,E2+1); T1 < E2, varf(H1+1,T1+1,E2)),
    (E1 < H2, varf(H2,T2,E1+1); T2 < E1, varf(H2+1,T2+1,E1)).
```

and then distribute the conjunction:

```
mq(E1,E2,T1,T2,H1,H2) ←
    phil(H1,E1,T1), phil(H2,E2,T2),
      T1 < E2, E1 < H2, varf(H1+1,T1+1,E2), varf(H2,T2,E1+1);
    phil(H1,E1,T1), phil(H2,E2,T2),
      E2 < H1, T2 < E1, varf(H1,T1,E2+1), varf(H2+1,T2+1,E1);
    phil(H1,E1,T1), phil(H2,E2,T2),
      T1 < E2, T2 < E1, varf(H1+1,T1+1,E2), varf(H2+1,T2+1,E1);
    phil(H1,E1,T1), phil(H2,E2,T2),
      E2 < H1, E1 < H2, varf(H1,T1,E2+1), varf(H2,T2,E1+1).
```

Next, we unfold (twice) the first `phil` constraint in the first alternative of the disjunction and the second `phil` constraint in the second alternative. Changes made in this step are shown in italics:

```
mq(E1,E2,T1,T2,H1,H2) ←
    H1 < E1, H1+1 <* E1+1, E1 < T1, E1+1 <* T1+1, T1 < H1+1,
      T1+1 <* H1+1+1, phil(H2,E2,T2),
      T1 < E2, E1 < H2, varf(H1+1,T1+1,E2), varf(H2,T2,E1+1);
    phil(H1,E1,T1), H2 < E2, H2+1 <* E2+1, E2 < T2, E2+1 <* T2+1,
      T2 < H2+1, T2+1 <* H2+1+1,
      E2 < H1, T2 < E1, varf(H1,T1,E2+1), varf(H2+1,T2+1,E1);
    phil(H1,E1,T1), phil(H2,E2,T2),
      T1 < E2, T2 < E1, varf(H1+1,T1+1,E2), varf(H2+1,T2+1,E1);
    phil(H1,E1,T1), phil(H2,E2,T2),
      E2 < H1, E1 < H2, varf(H1,T1,E2+1), varf(H2,T2,E1+1).
```

Folding the first and second alternatives, using the definition of `mq`, yields:

```
mq(E1,E2,T1,T2,H1,H2) ←
    T1 < E2, E1 < H2, H1 < E1, E1 < T1, T1 < H1+1,
      mq(E1+1,E2,T1+1,T2,H1+1,H2);
    E2 < H1, T2 < E1, H2 < E2, E2 < T2, T2 < H2+1,
      mq(E1,E2+1,T1,T2+1,H1,H2+1);
    phil(H1,E1,T1), phil(H2,E2,T2),
      T1 < E2, T2 < E1, varf(H1+1,T1+1,E2), varf(H2+1,T2+1,E1);
    phil(H1,E1,T1), phil(H2,E2,T2),
      E2 < H1, E1 < H2, varf(H1,T1,E2+1), varf(H2,T2,E1+1).
```

Now we unfold the third and fourth alternatives:

```
mq(E1,E2,T1,T2,H1,H2) ←
    T1 < E2, E1 < H2, H1 < E1, E1 < T1, T1 < H1+1,
      mq(E1+1,E2,T1+1,T2,H1+1,H2);
    E2 < H1, T2 < E1, H2 < E2, E2 < T2, T2 < H2+1,
      mq(E1,E2+1,T1,T2+1,H1,H2+1);
    H1 <* E1, E1 < T1, E1+1 <* T1+1, T1 < H1+1, T1+1 <* H1+1+1,
      H2 <* E2, E2 < T2, E2+1 <* T2+1, T2 < H2+1, T2+1 <* H2+1+1,
      T1 < E2, T2 < E1, varf(H1+1,T1+1,E2), varf(H2+1,T2+1,E1);
    H1 < E1, H1+1 <* E1+1, E1 < T1, E1+1 <* T1+1, T1 <* H1+1,
      H2 < E2, H2+1 <* E2+1, E2 < T2, E2+1 <* T2+1, T2 <* H2+1,
      E2 < H1, E1 < H2, varf(H1,T1,E2+1), varf(H2,T2,E1+1).
```

and then simplify the third and fourth alternatives, using the properties of ‘<’ given in (1) and (3) of Figure 1:

```
mq(E1,E2,T1,T2,H1,H2) ←
    T1 < E2, E1 < H2, H1 < E1, E1 < T1, T1 < H1+1,
       mq(E1+1,E2,T1+1,T2,H1+1,H2);
    E2 < H1, T2 < E1, H2 < E2, E2 < T2, T2 < H2+1,
       mq(E1,E2+1,T1,T2+1,H1,H2+1);
    time(E1,eternity), time(E2,eternity), time(T1,eternity), time(T2,eternity),
       time(H1+1,eternity), time(H2+1,eternity), H1 <* E1, H2 <* E2;
    time(E1,eternity), time(E2,eternity), time(T1,eternity), time(T2,eternity),
       time(H1,eternity), time(H2,eternity).
```

The fourth alternative, in which all variables are eternity events, is subsumed by each of the first two. It can be deleted by applying our fourth transformation rule:

```
mq(E1,E2,T1,T2,H1,H2) ←
    T1 < E2, E1 < H2, H1 < E1, E1 < T1, T1 < H1+1,
       mq(E1+1,E2,T1+1,T2,H1+1,H2);
    E2 < H1, T2 < E1, H2 < E2, E2 < T2, T2 < H2+1,
       mq(E1,E2+1,T1,T2+1,H1,H2+1);
    time(E1,eternity), time(E2,eternity), time(T1,eternity), time(T2,eternity),
       time(H1+1,eternity), time(H2+1,eternity), H1 <* E1, H2 <* E2.
```

We now define a new predicate `m2`, which is the same as `mq` except that one constraint is omitted from each of the first two alternatives:

```
m2(E1,E2,T1,T2,H1,H2) ←
    T1 < E2, H1 < E1, E1 < T1, T1 < H1+1,
       m2(E1+1,E2,T1+1,T2,H1+1,H2);
    T2 < E1, H2 < E2, E2 < T2, T2 < H2+1,
       m2(E1,E2+1,T1,T2+1,H1,H2+1);
    time(E1,eternity), time(E2,eternity), time(T1,eternity), time(T2,eternity),
       time(H1+1,eternity), time(H2+1,eternity), H1 <* E1, H2 <* E2.
```

By applying the result used in our unfold/fold transformation rule:

```
mq(E1,E2,T1,T2,H1,H2) → m2(E1,E2,T1,T2,H1,H2)
```

In the definition of `m2`, the third alternative is subsumed by each of the first two, so it can be deleted by applying our fourth transformation rule:

```
m2(E1,E2,T1,T2,H1,H2) ←
    T1 < E2, H1 < E1, E1 < T1, T1 < H1+1,
       m2(E1+1,E2,T1+1,T2,H1+1,H2);
    T2 < E1, H2 < E2, E2 < T2, T2 < H2+1,
       m2(E1,E2+1,T1,T2+1,H1,H2+1).
```

We now transform the definition of `ms`. By unfolding all of the constraints on the right side, distributing the conjunction, and then folding using the definition of `ms`, we obtain the following recursive definition:

```
ms(E1,E2,T1,T2,H1,H2) ←
    T1 < E2, H1 < E1, E1 < T1, T1 < H1+1,
       ms(E1+1,E2,T1+1,T2,H1+1,H2);
    T2 < E1, H2 < E2, E2 < T2, T2 < H2+1,
       ms(E1,E2+1,T1,T2+1,H1,H2+1).
```

which is the same as the definition of `m2`, modulo predicate renaming, and therefore `m2` and `ms` are equivalent.

To summarize, by transforming the definitions of `ms` and `mq` into equivalent self-recursive versions, and defining an intermediate predicate `m2`, we showed that

$$(q) \leftrightarrow \texttt{mq(E1,E2,T1,T2,H1,H2)} \rightarrow \texttt{m2(E1,E2,T1,T2,H1,H2)}$$
$$\leftrightarrow \texttt{ms(E1,E2,T1,T2,H1,H2)} \leftrightarrow (s)$$

This completes the proof that the Tempo query representing the shared variables algorithm implies the specification of the mutual exclusion problem.

4 Conclusions: Specifications vs Programs

This paper has demonstrated a way to derive concurrent algorithms within a new declarative programming language, Tempo. Here we briefly discuss the context into which Tempo fits.

Programming languages and specification languages both describe algorithms, but they are usually clearly distinguished: a programming language enables an algorithm to be executed efficiently by a real computer, while a specification language allows a human (or even a machine) to understand the algorithm and reason about it at an appropriate level of abstraction. It has been argued (Hayes and Jones, 1989) that this distinction is inevitable because the two types of language have different jobs to do and therefore have conflicting design objectives. However, history has shown that, faced with the need for two separate representations of an algorithm, programmers are likely to dispense with a formal specification and be satisfied with an informal justification of the executable program's correctness. Besides, the translation of a specification into a program remains a manual,[4] error-prone, process.

We believe that executable specification languages have an important role to play in program design. Although an executable specification may be a compromise between expressiveness and efficiency, the simplicity of a single representation of an algorithm is extremely appealing, especially for prototyping and pedagogical purposes. Tempo can be viewed as an executable specification language for concurrent systems, just as Prolog fulfils this function in its own application domain. A thorough comparison between Tempo and other concurrent languages is beyond the scope of this paper, and is the subject of ongoing research; however, we can make the following observations:

- As a programming language, Tempo has much in common with other concurrent symbolic languages such as Parlog, KL1, PCN, and April. The power of Tempo for writing real application programs is currently being tested, as features such as values are added to the language.
- As a specification language for concurrent systems, Tempo is much closer to languages based on temporal logic, e.g., Unity (Chandy and Misra, 1988) and TLA (Lamport, 1994), than process algebras (Milner, 1989). Both Unity and TLA can express the safety and progress properties of an algorithm, as well as (at an abstract level) the algorithm itself: in these languages an algorithm is correct if it logically implies the specification. This is true also in Tempo, except that a Tempo specification/algorithm can immediately be executed, and the progress

4 If it were automated, the specification language would effectively be executable!

properties are not explicit. An interesting aspect of TLA is that, for simplicity, it deliberately relies as far as possible on classical logic, resorting to temporal reasoning only where it is essential (mainly for progress properties). Tempo shares this aim, being based *entirely* on classical logic.

Acknowledgements

I am very grateful to John Gallagher for helpful advice on the content of this paper, and for filling in a 15-year gap in my knowledge of logic program transformation. Thanks are also due to the referees for comments on the presentation.

References

Boulanger, D. and Bruynooghe, M. 1993. Deriving unfold/fold transformations of logic programs using extended OLDT-based abstract interpretation. *Journal of Symbolic Computation 15*, pp. 495–521.

Chandy, K.M. and Misra, J. 1988. *Parallel Program Design*. Addison-Wesley.

Foster, I., Olson, R., and Tuecke, S. 1992. Productive parallel programming: The PCN approach. *Scientific Programming 1*, 1, pp. 51–66. Reprinted in *Programming Languages for Parallel Processing*, D. Skillicorn and D. Talia (Eds.), IEEE Computer Society Press, pp. 358–373, 1995.

Gregory, S. 1987. *Parallel Logic Programming in PARLOG*. Addison-Wesley.

Gregory, S. and Ramirez, R. 1995. Tempo: a declarative concurrent programming language. In *Proceedings of the 12th International Logic Programming Conference* (Tokyo, June 1995), L. Sterling (Ed.). MIT Press.

Hayes, I.J. and Jones, C.B. 1989. Specifications are not (necessarily) executable. *Software Engineering Journal 4*, 6, pp. 330–338.

Kowalski, R.A. 1979. Algorithm = Logic + Control. *Communications of the ACM 22*, 7, pp. 424–436.

Lamport, L. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems 16*, 3, pp. 872–923.

McCabe, F.G. and Clark, K.L. 1995. April: Agent Process Interaction Language. In *Intelligent Agents — Lecture Notes in Artificial Intelligence 890*. Springer-Verlag.

Milner, R. 1989. *Communication and Concurrency*. Prentice Hall.

Peterson, G.L. 1981. Myths about the mutual exclusion problem. *Information Processing Letters 12*, 3, pp. 115–116.

Proietti, M. and Pettorossi, A. 1994. Synthesis of programs from unfold/fold proofs. In *Proceedings of LOPSTR93* (Louvain-la-Neuve, July 1993), Y. Deville (Ed.). Springer-Verlag.

Ueda, K. and Chikayama, T. 1990. Design of the kernel language for the Parallel Inference Machine. *Computer Journal 33*, 6, pp. 494–500.