Tempo: a declarative concurrent programming language

Steve Gregory and Rafael Ramirez

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

steve@cs.bris.ac.uk ramirez@cs.bris.ac.uk

Abstract

Tempo is a declarative concurrent programming language based on classical first-order logic. It improves on traditional concurrent logic programming languages (e.g., Parlog) by *explicitly* specifying aspects of the behaviour of concurrent programs, namely their safety properties. This provides great advantages in writing concurrent programs and manipulating them while preserving correctness. The language has a procedural interpretation that allows the specification to be executed, also concurrently. Tempo is sufficiently high-level to simulate practical concurrent programming paradigms, and can act as a common framework in which algorithms for a variety of paradigms may be expressed, compared, and manipulated.

1 Introduction

With the ever more widespread use of multiprocessor architectures and computer networks, the study of concurrent programming is becoming increasingly important. At the same time, concurrent programming is substantially more difficult than sequential programming: as well as performing computation, processes must communicate and synchronize properly. For traditional sequential (or *transformational*) programs, we need only make sure that (a) the program's final result (if any) is correct, and (b) the program terminates. However, a concurrent (or *reactive*) program may need to satisfy various properties *during its execution*. These are classified into *safety* properties, those that must *always* be true, and *progress* (or *liveness*) properties, those that must *eventually* hold. The two properties of transformational programs noted above — partial correctness and termination — are special cases of safety and progress properties, respectively.

An example discussed in almost every textbook on concurrent programming (e.g., [3, 4]) is the mutual exclusion problem: Given a number of *philosopher* processes repeatedly executing two alternate sequences of instructions, a non-critical section (*thinking*) and a critical section (*eating*), allow no more than one process to execute its critical section at a time. The mutual exclusion requirement itself is a safety property; a desirable progress property is that a process trying to enter its critical section should eventually succeed (assuming that any process that enters its critical section eventually exits).

```
repeat
    think;    % Tb, Te
    hungry;   % H, E
    eat;      % Eb, Ee
    exit;     % T
forever
```

Figure 1: Mutual exclusion

The mutual exclusion problem is conventionally solved by defining an *entry protocol* (which we name *hungry*) and an *exit protocol*: additional code inserted before and after the critical section, as in Figure 1. While *think* and *eat* abstract the internal actions of a process, *hungry* and *exit* necessarily involve some form of communication between processes.

Numerous methods exist for interprocess communication and synchronization, varying according to their suitability for a particular architecture and their purported ease of use. For example, communication via shared variables may be suitable for a shared memory machine, while message passing is better adapted to distributed memory. Higher level "coordination languages" exist [6, 8], which may be easier to use but harder to implement efficiently.

To construct concurrent programs, or prove them correct, one must ensure that all required properties hold, regardless of the relative speed of the processes. Perhaps for this reason, concurrent programs require more effort to develop than sequential programs and tend to be less reliable. The proliferation of paradigms further complicates the situation, in that each may require a different algorithm to do a given job. We therefore believe it essential to seek easier ways to develop reliable concurrent software, preferably in a way which is not specific to any particular concurrent programming paradigm.

The ideal of declarative programming is that a program acts as its own specification, and therefore requires no separate correctness proof. As well as making programming easier, the benefits of this approach include the potential for program transformation, parallel execution, declarative debugging, and so on. In the case of Prolog, a transformational language, a program can be read declaratively as a description of the final result that it computes: provided that the program terminates, its result is guaranteed to satisfy this specification.

Concurrent logic programming languages such as Parlog [10] and KL1 [16] are declarative in the same sense as Prolog: a program can *explicitly* specify its final result. Unfortunately, all other (both safety and progress) properties of a program remain *implicit*: they have to be preserved by proper use of control features (modes, sequencing, etc.). Many concurrent logic programs (e.g., the mutual exclusion program in [15]) therefore have little or no declarative reading.

Overview of Tempo

For concurrent programming to benefit from the declarative approach, a program needs to explicitly describe more than just its final result. We propose a language in which *all safety properties* can be stated declaratively. Our language, Tempo, is based on classical first-order logic and has a procedural interpretation which allows specifications to be executed as concurrent programs.

Tempo explicitly describes processes (which we shall call *activities* to distinguish from computational processes) as partially ordered sets of atomic *events*. For example, consider a *thinker*: a simplified version of a philosopher (Figure 1) which goes through its think, hungry, eat, exit cycle only once before resuming thinking forever. We represent each of the four states by a pair of events denoting the beginning and end of the state, except the exit state which is represented by a single event; see the comments in Figure 1. A thinker is a totally ordered sequence of events, and can be represented in Tempo by a clause:

```
thinker(H,E,T) ← Tb < Te, Te < H, H < E, E < Eb, Eb < Ee, Ee < T.
```

We have written `H`, `E`, and `T` in the clause head to make them accessible to other constraints, providing the required synchronization with other thinkers. `Tb`, `Te`, `Eb`, and `Ee` are local events, representing internal actions. In the query

```
← thinker(H1,E1,T1), thinker(H2,E2,T2), (T1 < E2; T2 < E1).
```

the disjunction coordinates two thinkers, ensuring that the intervals `E1`,...,`T1` and `E2`,...,`T2` do not overlap; this in turn makes the thinkers' eat phases `Eb1`,...,`Ee1` and `Eb2`,...,`Ee2` mutually exclusive. The effect is the same as the program of Figure 1, but the relative order of events is entirely explicit. If we do not need to represent local events, the `thinker` clause could be written more abstractly as

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

Related work

Concurrent logic programming is one major source of inspiration for this work. Another is Unity [7], a simple concurrent programming language specially designed so that temporal properties of a program's behaviour can easily be proven in an accompanying temporal logic. Unity is also intended to be able to express algorithms for a variety of paradigms. Our work has similar objectives, the major difference being that Tempo is a declarative language.

There are several languages based on an executable temporal logic, e.g., [1, 9, 12]. Like Tempo, these languages allow temporal properties of a program to be represented explicitly. However, we are not aware of a language aimed at expressing concurrent algorithms in a general way. Concurrent Metatem [9] is perhaps the closest, but that language makes a clear distinction between processes, expressed in temporal logic, and the interface between them, which is handled by a specific message passing mechanism.

Sections 2 to 5 introduce the basic features of Tempo, showing how it can describe the behaviour of processes. Section 6 explains how such a specification can be executed concurrently. Section 7 illustrates the use of Tempo, including the derivation of concurrent algorithms from a Tempo specification. Section 8 summarizes the language and its applications, and some areas of future work.

2 Time, events, and precedence

Many researchers, e.g., [11, 14], have proposed methods for reasoning about concurrency or other temporal phenomena using partially ordered sets of events. As hinted above, Tempo is based on the same general idea. The event ordering relation $X < Y$, read "X precedes Y", is the *only* primitive predicate in Tempo: its domain is the set of events. Events in Tempo are atomic, so we can define an event to be a value representing the time at which the event is executed (in Section 4 we redefine events to include the time value as one of several attributes). The set of time values T could be any infinite set (e.g., the set of real numbers or of natural numbers), augmented by the value `eternity`, which is

```
∀X∀Y∀Z (X < Y ∧ Y < Z → X < Z)          (1)
∀X (X < eternity)                        (2)
∀X (X < X → X = eternity)                (3)
∀X (eternity < X → X = eternity)         (4)
```

Figure 2: Properties of the ‘<’ predicate

interpreted as the “end of time”. Figure 2 defines the ‘<’ predicate ((4) is a corollary of (1–3)): ‘<’ is transitive and nothing precedes itself except `eternity`, which is preceded by every member of the domain.

Consider a first-order language with no function symbols, and only one predicate symbol, ‘<’, though we shall relax the latter restriction later. Syntactically, no constants are allowed in queries or clauses. A *query* in such a language is then a definite goal comprising a conjunction of *precedence constraints* between variables. An implementation computes a value from *T* for each variable in the query. For example (upper case identifiers are variables):

```
← A < B, A < C.
```

has an infinite number of answer substitutions, including `{A/1, B/3, C/2}` or `{A/1, B/2, C/2}`, but not `{A/1, B/2, C/1}`. The thinker activity of Section 1 can be described by (Q1), whose solution is any substitution such that `H < E < T`:

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

Our prohibition of constants from the language syntax means that a query cannot specify the *absolute* values of variables, *except* that a variable `X` can be given the value `eternity` by a constraint such as `X < X`; this follows from (3) of Figure 2. For example, the query

```
← A < B, A < C, A < A.
```

has a single solution: `{A/eternity, B/eternity, C/eternity}`.

Execution

This simple language is a subset of Tempo. However, the key point about Tempo is that its procedural interpretation actually *executes* the specified activities, rather than merely returning an answer substitution for query variables:

1. Instead of computing a substitution `E/t`, Tempo executes an event (which we shall refer to by the variable name `E`) at time `t`. No event is executed for a variable that has the value `eternity`. 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.

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.

Figure 3 shows an interpreter for this subset of Tempo. It uses a constraint set (*CS*), containing the constraints still to be satisfied, and a try list (*TL*), a FIFO queue containing the events (variables) that have not yet been executed but need to be tried. A trace of the interpreter for query (Q1) is shown in Figure 4.

```
place query constraints in constraint set CS;
place query variables in try list TL;
while TL not empty:
    delete first member E of TL;
    if E is enabled then execute E.

execute E:   for each precedence constraint in CS with E on left, E<R:
                 delete E<R;
                 add R to rear of TL if not in TL.

E is enabled if:  E does not appear on right of a precedence constraint in CS.
```

Figure 3: Simple Tempo interpreter

CS = {`H<E`, `E<T`}	*TL* = [`E`,`H`,`T`]	`E` is not enabled
	TL = [`H`,`T`]	`H` is enabled: execute `H`
CS = {`E<T`}	*TL* = [`T`,`E`]	`T` is not enabled
	TL = [`E`]	`E` is enabled: execute `E`
CS = { }	*TL* = [`T`]	`T` is enabled: execute `T`
	TL = []	terminate

Figure 4: Trace of Tempo interpreter for thinker example

3 User-defined constraints

A constraint definition is a single definite clause $H \leftarrow B$ where *H* (the *head*) is an atomic formula and *B* (the *body*) is a conjunction of (primitive or user-defined) constraints. *H*, and each constraint in *B*, must have one or more arguments, all variables; the arguments of *H* must be distinct (because we do not implement equality).

For example, given the `thinker` constraint defined in Section 1, the query

```
← thinker(H,E,T).
```

is equivalent to query (Q1). The following query represents a clock that repeatedly ticks for ever, yielding an infinite execution sequence "*execute* `Tick`; *execute* `Tick1`; *execute* `Tick1'`; ...":

```
← clock(Tick).

clock(Tick) ← Tick < Tick1, clock(Tick1).
```

It can be shown that every constraint, whether primitive or user-defined, is true if all of its arguments are `eternity`, so every query has a trivial solution in which all variables have the value `eternity`, i.e., no events are executed!

Execution

When trying an event `E`, any user-defined constraints in which `E` appears are *expanded*, until `E` can be determined to be either disabled (it appears on the right of a '<' constraint) or enabled (it does not appear except possibly on the left of a '<' constraint). While expanding constraints, *body variables* (like `Tick1` in the `clock` clause above) are scheduled by adding them to the try list.

Termination

With user-defined constraints, the interpreter may not terminate, either because of a circular constraint definition (e.g., `p(E) ← p(E)`), or because body variables may be repeatedly added to the try list, as in the clock example above.

An infinitely-defined constraint need not cause non-termination: a constraint *C* is not expanded if all of its arguments appear on the right of '<' constraints. The justification for this is that (a) *C* must be true and (b) it cannot further constrain the query variables in *C*, since these already have the value `eternity`. E.g., the following query immediately terminates without executing any events:

```
← clock(Tick), Tick < Tick.
```

4 Event trees: long-lived activities

Most interesting activities are long-lived: they comprise a large or infinite set of events, which it is not feasible to name individually. The clock example suggests a way to handle such cases — representing events by body variables — but these variables are not accessible from the query or from other constraints. Another possibility would be to define constraints on data structures of events, e.g.:

```
clock([T,T1|Ticks]) ← T < T1, clock([T1|Ticks]).
```

but then different constraints could have different types (event, list, tree, etc.), and would be difficult to compose.

The solution that we adopt in Tempo is to allow an event to be associated with one or more other events: its *offsprings*. The offsprings of `E` are named `E+1`, `E+2`, etc., and are implicitly preceded by `E`, i.e., `E < E+`*N*, for all *N*. Syntactically, offspring names are allowed in queries and in bodies (but not heads) of constraint definitions. These restrictions mean that an event can be enabled as soon as it no longer appears in the constraint set, thus simplifying the interpreter.

An improved definition of a clock is shown in Figure 5(a). Figure 5(b) defines the philosopher of Figure 1: successive transitions from thinking to hungry state are named by a sequence of events `H`, `H+1`, `H+1+1`, etc., and likewise for events `E` and `T`. Event trees need not be lists: e.g., Figure 5(c) defines a pigeon, which is hatched (`H`), has two offsprings (`H+1`, `H+2`), and then dies (`H+3`).

```
clock(Tick) ← clock(Tick+1).                                         (a)

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

E <* F ← E < F, E+1 <* F+1.

pigeon(H) ← pigeon(H+1), pigeon(H+2), H+1 < H+3, H+2 < H+3.          (c)
```

Figure 5: Event trees — (a) clock; (b) philosopher; (c) pigeon

Logical meaning

An event is no longer a time value, but instead has a time value associated with it. We assume two selector relations:

`time(E,T)`: event `E` has time value `T`;
`offs(E,N,C)`: event `C` is the *N*th offspring of `E`.

The meaning of the '<' predicate can now be defined as:

$$\forall E_1 \forall E_2\ (E_1 < E_2 \leftrightarrow \exists T_1 \exists T_2\ (\mathtt{time}(E_1,T_1) \wedge \mathtt{time}(E_2,T_2) \wedge T_1 <_s T_2))$$

where '$<_s$' is a new name for the predicate defined in Section 2.

A constraint definition containing event names of the form `E+N` is not itself a clause, but it can easily be translated to one. For each event `E+N`, we replace `E+N` by a new variable `EN`, and add two conditions `offs(E,N,EN), E < EN` to the body. For example, the logical meaning of the constraint of Figure 5(b) is:

```
phil(H,E,T) ← offs(H,1,H1), H < H1, H <* E, E <* T, T <* H1.
```

Execution

Operationally, events named `E+N` are treated exactly like body variables. Indeed, our interpreter needs no modification to handle event trees, provided that all implicit `E < E+N` constraints are added explicitly to constraint definitions. For example, in Figure 5(a), `Tick+1` acts like a simple body variable that is preceded by `Tick`. The difference is that `Tick+1` can be accessed by other constraints that share variable `Tick`, as illustrated in query (Q2) below.

Termination

As explained in Section 3, a constraint all of whose arguments have the value `eternity` will not be expanded. No offsprings will be generated for an event `E` appearing in such a constraint even if, logically, offsprings exist. The justification for this is that the time values of all descendants of `E` are known to be `eternity`, because event trees are ordered. For example, query (Q2) defines a clock that ticks three times and then stops; although `Tick` is an infinite tree, only part of the tree is generated, as shown in Figure 6.

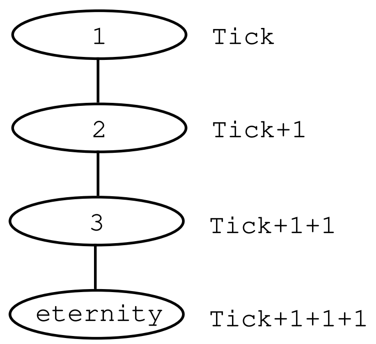

Figure 6: Solution to query (Q2)

```
← clock(Tick), Tick+1+1+1 < Tick+1+1+1.
```
(Q2)

The following example defines a terminating activity, a thinker, in terms of a non-terminating one, a philosopher:

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

5 Disjunction

In general, we may need to define a constraint by multiple clauses. Since the head arguments are distinct variables, all clause heads can be made identical, and a set of clauses written as a single clause containing a disjunction (the disjunction operator '`;`' is less tightly binding than '`,`' but more tightly binding than '←'):

$$H \leftarrow Cs_1;\ \ldots;\ Cs_n.$$

We therefore retain the restriction to single-clause definitions, but each constraint may be either atomic or a disjunction of conjunctions `(`Cs_1`; ...; `Cs_n`)`.

For example, Figure 7 defines mutual exclusion between two philosophers.

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

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

Figure 7: Mutual exclusion between two philosophers

Execution

In the absence of disjunction, a query determines a unique set of constraints. A Tempo interpreter produces *any* execution sequence satisfying those constraints; we don't care which one. With disjunction, a single set of constraints must be chosen from among many possible sets: *one* alternative must be selected from each disjunction. A key feature of Tempo is the way that this is done, as follows.

While testing whether an event `E` is enabled, every alternative of each disjunction constraint *D* is tried. If `E` is enabled by some but not all alternatives of *D* (we say `E` is *conditionally enabled*), we enable `E` and *reduce D*: retain the enabling alternatives and delete the others. If all alternatives of *D* enable `E`, `E` is enabled; if no alternatives enable `E`, `E` is disabled — in either case, *D* is not reduced. For example, given the query

```
← B < A, C < A, p(A); A < B, C < B, q(B); A < C, B < C, r(C).
```

`A` could be enabled by reducing the disjunction to

```
← C < B, q(B); B < C, r(C).
```

and then `C` could be enabled by reducing the disjunction to a single alternative

```
← q(B).
```

A different alternative will be chosen depending on which of the three events is the last to occur. (A more obvious way to handle disjunction would be, if one or more alternatives enable `E`, to enable `E` and *commit* to any one of the enabling alternatives. The problem is that commitment would be determined by a single event. In the above example, `A` would be enabled by committing to *either* the second or third alternative of the disjunction.)

To avoid premature commitment, disjunctions that conditionally enable `E` are reduced only if `E` is enabled by all other (non-disjunctive) constraints. For example, in the program of Figure 7, the `mutex(E1,E2,T1,T2)` constraint will first be expanded to the disjunction

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

If the `phil(H1,E1,T1)` constraint enables `E1`, then `E1` can be enabled by deleting the second alternative of the disjunction, reducing it to:

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

If, instead, the `phil(H2,E2,T2)` constraint first enables `E2`, we would enable `E2` and delete the first alternative of the disjunction. The alternative chosen depends on which of `E1`, `E2` is first enabled by the other constraints in the query.

Programming considerations

For every disjunction there must be at least one event, say `E`, that is enabled in some alternatives and disabled in others; otherwise the disjunction would never be reduced. In general, there may be many such events, including (say) `F`. The disjunction will be reduced as soon as *any* of these events is tried. If the programmer does not want the disjunction to be reduced when `F` is tried, he can simply add a `E < F` constraint outside the disjunction.

To simplify implementation, we prevent user-defined constraints from being expanded inside a disjunction. Each event `E` appearing in a user-defined constraint in a disjunction is implicitly preceded by the event that causes commitment to the alternative in which it appears. For example, in the `mutex` disjunction of Figure 7, `E1` and `E2` cause commitment to the first and second alternatives, respectively, so the disjunction is effectively translated to:

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

All of these implicit constraints are subsumed by the explicit constraints in the query of Figure 7, so the meaning of the program is not affected.

6 Concurrency

We can parallelize our Tempo interpreter by allowing a query to be solved cooperatively by a number of *workers*. Each worker has its own constraint set and try list, and solves the constraints allocated to it in essentially the same way as a sequential interpreter. The main difference is that an event cannot be executed until it has been enabled by *all* workers that share it.

The allocation of constraints to workers could be automatic. However, as we are interested in expressing concurrent programs, we provide a syntactic way for the programmer to explicitly structure a program into *processes*: sets of constraints, each allocated to a different worker.

A query may take the form

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

where each Cs_i is a conjunction of constraints which is executed as a separate process. A constraint definition may take the form

$$H \leftarrow C_1,\ \ldots,\ C_n,\ \{Cs_1\},\ \ldots,\ \{Cs_m\}.$$

When the constraint is expanded, constraints C_1, ..., C_n are added to the current process, and m new processes are spawned to solve $\{Cs_1\}$, ..., $\{Cs_m\}$.

Execution

A concurrent interpreter for Tempo works essentially like a sequential one, but with two main differences:

1. When a $\{Cs_i\}$ constraint is encountered a new invocation of the interpreter is spawned, with Cs_i in its constraint set and the variables of Cs_i in its try list.
2. When an event `E` is (unconditionally) enabled in a process, that process

marks `E` as enabled, instead of immediately executing it. The process will execute `E` when it has been enabled by all other processes; meanwhile, it will proceed with enabling and executing other events.

The handling of disjunction is a little more complex in the concurrent interpreter. An event is conditionally enabled if it is conditionally enabled in one or more processes and unconditionally enabled in every other process that shares it. Therefore, if `E` is conditionally enabled in a process, this process cannot simply mark `E` as enabled and reduce its disjunctions, because `E` may be disabled in some other process; neither can it wait for all other processes to mark `E` as enabled, as this would result in deadlock if `E` is conditionally enabled in more than one process. There are two possible solutions:

1. When an event `E` is conditionally enabled in a process *P*, *atomically* check all other processes that share `E` but have not marked it as enabled. If they can all enable `E`, reduce disjunctions in all of the processes, otherwise treat `E` as disabled. This method, which is comparable to that required for synchronous message passing languages with output guards [3, 5], can be expensive in a parallel implementation.
2. Restrict the language so that each event `E` can be conditionally enabled in at most one process. If `E` is conditionally enabled in *P*, *P* waits for all other processes to mark `E` as enabled before enabling it. We adopt this restriction in the remainder of the paper.

7 Concurrent programming in Tempo

One way to develop algorithms in Tempo is by transforming a specification *S* to a program that logically implies *S*. The derived program is guaranteed to have the same safety properties as the specification, though their progress properties may differ; e.g., one may terminate and the other not. Finally, the program can be turned into a *concurrent* one by grouping constraints into processes. This final step affects neither the safety nor the progress properties of the algorithm, provided that the restrictions of Section 6 are observed.

As a concurrent programming language, Tempo appears to add to the proliferation of paradigms mentioned in Section 1: processes communicate via a new medium, *shared events*. However, our objective is rather to simplify matters by providing a framework in which algorithms for a variety of paradigms can be expressed, derived, and compared. In this section we illustrate this by showing the systematic derivation in Tempo of mutual exclusion algorithms for three different concurrent programming paradigms.

Synchronous message passing

Any Tempo program in which each event is shared by at most two processes simulates a program using point-to-point synchronous message passing. The shared event represents a message exchange between a sender and receiver, which takes place when both are ready (both have enabled the event). If the event is conditionally enabled in one process, that is the receiver, otherwise there is no distinction. *Local* (unshared) events represent a process's internal actions.

Figure 7 is an (executable) specification of mutual exclusion between two philosophers. We can turn this into a concurrent program simply by allocating each constraint in the query to a separate process, as shown in Figure 8(a). Here, `H1` and `H2` are local events; `E1`, `E2`, `T1`, and `T2` are shared events, corresponding to synchronous message exchanges between the first two processes (the senders) and the third process (the receiver).

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

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

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

Figure 8: Tempo programs for mutual exclusion — (a) synchronous message passing; (b) asynchronous message passing; (c) shared mutable variables

```
% {phil(H1,E1,T1)}                % {mutex(E1,E2,T1,T2)}          (a)
process client1:                  process server:
repeat                            repeat
   think;                            client1?eat (E1) →
   (H1)                                 client1?think (T1)
   server!eat (E1);               or client2?eat (E2) →
   eat;                                 client2?think (T2)
   server!think (T1);             forever
forever
```

```
% {phil(H1,Eb1,T1), E1 <* Eb1}    % {H1 <* E1, H2 <* E2,
process client1:                  %  mutex(E1,E2,T1,T2)}          (b)
repeat                            process server:
   think;                         repeat
   server!hungry (H1);               client1?hungry (H1) →
   server?eat (E1);                     client1!eat (E1);
   eat (Eb1);                           client1?think (T1)
   server!think (T1);             or client2?hungry (H2) →
forever                                 client2!eat (E2);
                                        client2?think (T2)
                                  forever
```

```
Initially: b1 := false; b2 := false;                               (c)
% {phil(H1,E1,T1)}                % {phil(H2,E2,T2)}
process client1:                  process client2:
repeat                            repeat
   think;                            think;
   b1 := true (H1);                  b2 := true (H2);
   loop until not b2 (E1);           loop until not b1 (E2);
   eat;                              eat;
   b1 := false (T1);                 b2 := false (T2);
forever                           forever
```

Figure 9: Mutual exclusion solutions — (a) synchronous message passing; (b) asynchronous message passing; (c) shared mutable variables

Figure 9(a) shows the same algorithm (the first and third processes in the query of Figure 8(a)) in a conventional imperative style: the notation `proc?message` used here means "receive `message` from process `proc`", while `proc!message` means "send `message` to process `proc`".

Asynchronous message passing

Asynchronous message passing can be simulated by a Tempo program in which each shared event appears on the right of a '<' constraint in exactly one process: the sender. Other processes sharing the event are receivers. The shared event then corresponds to the asynchronous sending of a message: it can be delayed only by the single sender process.

We can derive an asynchronous program from Figure 7 by strengthening the specification (interposing a new local event `Eb1` between `E1` and `T1`, and `Eb2` between `E2` and `T2`) and dividing the query into processes, as in Figure 8(b). Now `H1`, `H2`, `E1`, `E2`, `T1`, and `T2` represent asynchronous messages. Figure 9(b) shows an analogous imperative program (for the first and third processes).

Shared mutable variables

Tempo can also model programs in which processes communicate via shared mutable variables (variables that can be destructively modified and read by several processes). Each shared variable is represented by a `var` constraint, contained in its own process, and events can be shared only between a `var` process and another process.

A `var` constraint defines a relation between two types of event: assignments to a variable and observations of the variable's value. 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. Suppose that assignments to true and false strictly alternate, beginning with an assignment to true, and we only want to observe the false values. Then the variable can be represented by a simpler constraint `varf(AT,AF,OF)`, defined in Figure 10.

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

Figure 10: Mutable variable constraint

Figure 8(c) is a mutual exclusion program for this paradigm, analogous to the imperative program of Figure 9(c). As this query implies that of Figure 7, mutual exclusion is ensured.

Concurrent programming issues

Fairness. Tempo has an implicit guarantee of *fairness*: every event that is enabled will eventually be executed. In a sequential interpreter, this can be implemented by making the try list a FIFO queue.

Deadlock. Deadlock and termination are conceptually identical in Tempo: they both occur when all remaining events have the value `eternity`. All that distinguishes deadlock is that `E = eternity` is implied by constraints spread across two or more processes. For example, in the program of Figure 8(c), if `H1` and `H2` occur at similar times, the processes become:

```
← {..., E1 < T1, T1 < H1+1, ...}, {..., E2 < T2, T2 < H2+1, ...},
  {..., T1 < E2, ...}, {..., T2 < E1, ...}
```

It follows that `E1`, `E2`, `T1`, `T2`, `H1+1`, `H2+1` are `eternity` and so no more events will ever be executed. Incidentally, it is quite easy to implement a deadlock-free algorithm such as that of Peterson [13] in Tempo.

Priority. Figure 11(a) shows a solution to the classic readers and writers problem: a `mutex` constraint holds between each writer and every other (reader or writer) process, but not between readers. Here, readers have (weak) precedence over writers because any hungry reader can eat immediately, while a hungry writer has to wait until all readers are idle. To give writers priority, a `prior` constraint is imposed between each writer and each reader, as in Figure 11(b): if a writer *W* becomes hungry before a reader *R*, then *W* will eat before *R*. `H` is no longer a local event: it represents a synchronous message from a `phil` process to the `mutex`/`prior` process.

```
← {phil(Hw,Ew,Tw)},                              % 1 writer    (a)
  {phil(Hr1,Er1,Tr1)}, {phil(Hr2,Er2,Tr2)},      % 2 readers
  {mutex(Ew,Er1,Tw,Tr1), mutex(Ew,Er2,Tw,Tr2)}.

← {phil(Hw,Ew,Tw)},                              % 1 writer    (b)
  {phil(Hr1,Er1,Tr1)}, {phil(Hr2,Er2,Tr2)},      % 2 readers
  {mutex(Ew,Er1,Tw,Tr1), mutex(Ew,Er2,Tw,Tr2),
   prior(Hw,Hr1,Ew,Er1), prior(Hw,Hr2,Ew,Er2)}.

prior(H1,H2,E1,E2) ←
  H1 < H2, E1 < E2, prior(H1+1,H2,E1+1,E2);
  H2 < H1, prior(H1,H2+1,E1,E2+1).
```

Figure 11: Readers and writers — (a) reader priority; (b) writer priority

8 Conclusions

Tempo is a concurrent programming language which is declarative in that a program is both an algorithm and a specification of the algorithm's safety properties. Its syntax is extremely simple, with no connectives other than conjunction and disjunction.

Conjunction is the sole means of program composition. It is used (a) to combine several processes into one program, and *vice versa*, and (b) to modify the behaviour of a process in an "additive" way: for example, to introduce new events (which can then be used by other processes) or even to make a non-terminating process terminate. These two forms of composition have been called *union* and *superposition*, respectively [7]. The advantage of Tempo is that superposition (as well as union) can be achieved without changing the code defining the process.

A key feature of Tempo is the way that disjunction is handled. A hypothetical theorem prover might generate a single execution sequence satisfying a given query by performing a complete search over its constraints. However, as well as being very expensive (impossible for a infinite set of constraints), no events could be executed until a complete solution is obtained. As a *programming* language, Tempo trades completeness for efficiency and the ability to execute events incrementally.

Tempo can be used to develop concurrent algorithms for a particular target paradigm, as demonstrated in Section 7. First, the problem is specified in a perspicuous manner in Tempo. Then the specification is strengthened until the constraints can be divided into processes that communicate using the intended target paradigm. If desired, local events can be introduced to represent internal actions. The resulting program can then be executed in Tempo, or mapped into a conventional concurrent language based on the target paradigm. Conversely, it is also possible to translate an existing concurrent algorithm for a particular paradigm into Tempo, and then prove properties of the algorithm by showing that the Tempo program implies a specification in Tempo.

Current status

A prototype, unoptimized, implementation of the complete Tempo language has been written in Parlog, and used to test all of the examples in this paper. It is available from `http://star.cs.bris.ac.uk`.

Future work

Giving events duration. Instead of events being *atomic*, they could be treated as time intervals during which other events can occur [2, 11]: those events can be further decomposed to provide an arbitrary degree of detail. This could be useful in deriving programs from specifications: if event `F` is included within `E`, `F` will automatically inherit all of `E`'s predecessors and successors. To do this in Tempo, we could give each event a *beginning* and *end* time value, and define four primitive precedence constraints: "`E` ends before `F` begins" (the existing '<'), "`E` begins before `F` begins", "`E` ends before `F` ends", "`E` begins before `F` ends".

Equality. Although mutually unconstrained events can occur simultaneously, there is currently no way to *force* two events to be equal. The examples that we have shown do not require this, but it should be easy to implement an equality constraint.

Real time. We have expressly excluded constants from the syntax of Tempo. Allowing constant time values to appear as arguments of constraints would yield a real-time programming language.

Progress. Safety properties of a Tempo program are explicit, but progress properties remain implicit. It would be desirable to be able to express these explicitly in a program, but so far we have not devised a way to do this.

Programming language features. We believe that Tempo can act as a basis for both imperative and declarative kinds of practical programming language, but the language currently lacks practical features such as numbers and data structures, and operations on them. We are currently working on the design of complete programming languages that combine Tempo with such features, especially from logic programming: this would produce a fully declarative concurrent programming language. Space prevents further discussion of these ideas, which will be presented in a future paper.

Acknowledgements

We are grateful to Keith Clark, David Warren, and the referees for comments on the presentation. Rafael Ramirez is supported by a scholarship from UNAM (National University of Mexico).

References

[1] Abadi, M. and Manna, Z. Temporal logic programming. *Proc. Symposium on Logic Programming* (San Francisco, August), S. Haridi (Ed.), IEEE Computer Society Press, pp. 4–16, 1987.

[2] Allen, J.F. Maintaining knowledge about temporal intervals. *Comm. ACM 26*, 11, pp. 832–843, 1983.

[3] Andrews, G.R. *Concurrent Programming: Principles and Practice*. Benjamin/ Cummings, 1991.

[4] Ben-Ari, M. *Principles of Concurrent and Distributed Programming*. Prentice Hall, 1990.

[5] Bernstein, A.J. Output guards and non-determinism in CSP. *ACM Trans. on Programming Languages and Systems 2*, 2, pp. 234–238, 1980.

[6] Carriero, N. and Gelernter, D. How to write parallel programs: a guide to the perplexed. *ACM Computing Surveys 21*, 3, pp. 323–357, 1989.

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

[8] Chandy, K.M. and Taylor, S. *An Introduction to Parallel Programming*. Jones and Bartlett, 1991.

[9] Fisher, M. Concurrent Metatem — a language for modelling reactive systems. *Proc. Parallel Architectures and Languages Europe* (Munich, June), 1993.

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

[11] Kowalski, R.A. and Sergot, M.J. A logic-based calculus of events. *New Generation Computing 4*, pp. 67–95, 1986.

[12] Moszkowski, B. *Executing Temporal Logic Programs*. Cambridge University Press, 1986.

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

[14] Pratt, V. Modeling concurrency with partial orders. *International Journal of Parallel Programming 15*, 1, pp. 33–71, 1986.

[15] Shapiro, E.Y. The family of concurrent logic programming languages. *ACM Computing Surveys 21*, 3, pp. 413–510, 1989.

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