A Declarative Approach to Concurrent Programming

Steve Gregory

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

steve@cs.bris.ac.uk

Abstract

Tempo is a language based on classical first-order logic, which can be used for both specifying and implementing concurrent systems. As a declarative programming language, it is unique in that a program explicitly specifies the safety properties that it satisfies. Unlike most specification languages, a Tempo specification of a concurrent system can be executed, also concurrently.

An earlier paper presented a subset of Tempo, in which simple synchronization problems could be expressed by precedence relations between events. In this paper we describe a major extension to the language which greatly increases its expressive power. In particular, the language now includes facilities to perform computation — data values, operations on them, and control features — as well as synchronization.

1 Introduction

1.1 Background

Concurrent and distributed software systems are becoming increasingly familiar, with the widespread use of computer networks and multiprocessor architectures. However, concurrent (reactive) programming remains a more difficult task than writing sequential (transformational) programs. One reason is the extra programming effort required for interprocess synchronization and communication; another is that the sequence of actions executed by a concurrent program is inherently non-deterministic, depending on the relative speeds of processes, and the program must behave correctly in *any* possible execution. A further complication is that numerous different concurrent programming paradigms exist. To solve a problem efficiently for a new paradigm may require a different algorithm; at best, the same algorithm may need to be expressed in a different way.

We believe that it is important to find ways to ease the task of constructing correct and reliable concurrent software. The approach that we advocate is to provide a high-level programming language in which there is a clear connection between a concurrent program and the properties that the program satisfies. This can not only make it easier to write a program that satisfies its specification, but can also expose the relationship between programs written for different paradigms. Before describing the language that we propose, we briefly discuss the background to this research.

Specifications and Programs. One method of improving the reliability of programs is by proving that a program satisfies a specification, or systematically deriving a program from such a specification. The design of languages for specifying concurrent systems is an active research area (Section 4.2). They can usually express the *safety* properties (those that always hold) and *progress* properties (those that eventually hold) of concurrent algorithms. Unfortunately, no known specification language has been able to bridge the gap between specifications and programs written in real, widely used programming languages: proving or deriving such programs remains a manual process, with no guarantee of correctness.

Declarative Programming. A declarative program can, ideally, be viewed as both an executable program and a specification of the program's behaviour. We believe that declarative languages have an important role to play in program design. In many circumstances (e.g., for pedagogical and prototyping purposes) it is impractical to maintain two separate representations (program and specification) of an algorithm. Then an executable specification can offer the "best of both worlds".

In practice, declarative languages have tended to be used more for the features that they provide than because they are declarative. For example, concurrent logic programming languages such as Parlog (Gregory, 1987) and KL1 (Ueda and Chikayama, 1990) are symbolic concurrent programming languages which can be implemented quite efficiently. However, they are declarative only in that a program explicitly describes the final result that it computes (not a very important property in most concurrent programs). All other safety and progress properties are *implicit*, relying on the proper use of control features, so they are therefore not necessarily preserved by program transformation or parallel execution.

1.2 Tempo

Tempo is a concurrent programming language which has much in common with Parlog, KL1, etc., but is significantly more declarative: a Tempo program explicitly describes *all safety properties* that it satisfies, not just the final result that it computes, so these properties are preserved by transformation, parallel execution, etc.

Like some other formalisms (Pratt, 1986; Kowalski and Sergot, 1986), Tempo describes concurrent systems as partially ordered sets of atomic *events*. Tempo differs from those languages by possessing a procedural interpretation that allows these descriptions to be executed. For example, a semaphore could be represented in Tempo by a relation between `P` and `V` (sequences of events representing decrementing and incrementing the semaphore, respectively) and `N`, the semaphore's initial value. The *temporal constraint* `semaphore(P,V,N)` simply states that, at any time, the number of `P` events executed less the number of `V` events executed cannot exceed `N`:

```
semaphore(P, V, N) ← V«P, semaphore(P, V#1, N+1);
                     P«V, N>=1, semaphore(P#1, V, N-1).
```

Tempo was originally presented in (Gregory and Ramirez, 1995). That version of the language could express the synchronization aspects of a limited range of examples, but could not perform computation. The new language presented here is a major extension to, and a superset of, the original language: it is able to represent and manipulate data values (numbers, data structures, etc.), as required in a general purpose programming language. This has been achieved in a declarative framework and without modifying the original subset of the language.

1.3 This Paper

Section 2 describes the design of Tempo. The language features are presented incrementally, beginning with the original subset of Tempo and successively relaxing restrictions on the syntax. For each extension, the syntax and operational semantics are stated, justified, and illustrated. The Tempo language itself is simple and based on a familiar logical syntax, but the procedural interpretation is new, so the emphasis is on explaining how the language can be implemented. We assume an implementation comprising a "compiler", which adds constraints to the source program, and an interpreter. The paper presents a simple interpreter and lists the compilation rules. Section 3 demonstrates the expressive power of Tempo by showing how a few typical concurrent problems can be solved. Section 4 summarizes Tempo, compares it with other concurrent languages, and outlines future research.

2 The Tempo Language

2.1 Events and Precedence

The key feature of Tempo is the *temporal constraint*, which describes a relation between *events*. Events are atomic (they have no duration), so their main attribute is their *execution time*. A temporal constraint is an atomic formula $p(t_1,...,t_j)$, where p is a *temporal relation*, either primitive or defined by a Tempo program.

The primitive temporal relation '«' (see Figure 1) constrains the order in which two events are executed: X«Y is read "X precedes Y" and indicates that the execution time of event X is earlier than that of 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 is the special value *eternity*; in this case, X is an *eternity event*. Eternity events are *never* executed. The '«' predicate is transitive and no event precedes itself except an eternity event, which is preceded by *every* event.

$$\forall X \forall Y \forall Z \ (X \ll Y \ \wedge \ Y \ll Z \ \rightarrow \ X \ll Z)$$
$$\forall X \forall Y \ (\mathit{time}(Y, \mathit{eternity}) \ \rightarrow \ X \ll Y)$$
$$\forall X \ (X \ll X \ \rightarrow \ \mathit{time}(X, \mathit{eternity}))$$
$$\forall X \forall Y \ (\mathit{time}(Y, \mathit{eternity}) \ \wedge \ Y \ll X \ \rightarrow \ \mathit{time}(X, \mathit{eternity}))$$
$$p(s_1, ..., s_i, X\#N, t_1, ..., t_j) \ \leftrightarrow \ (\exists Y \ \mathit{offs}(X, N, Y) \wedge X \ll Y \wedge p(s_1, ..., s_i, Y, t_1, ..., t_j))$$
$$Y = X\#N \ \wedge \ Z = X\#N \ \rightarrow \ Y = Z$$
$\mathit{offs}(X, N, Y)$: event Y is the Nth offspring of event X.

Figure 1: Properties of '«' and '#'

Tempo programs define each temporal relation r by a single definite clause whose head arguments are distinct variables V_1, ..., V_k:

$$r(V_1, \ ..., \ V_k) \ \leftarrow \ C_1, \ \ ..., \ \ C_m.$$

This is logically equivalent to

$$r(V_1, \ ..., \ V_k) \ \leftrightarrow \ \exists V_B \ \ C_1 \ \wedge \ ... \ \wedge \ C_m.$$

where V_B denotes the clause's *body variables*: those appearing in C_1, ..., C_m but not in V_1, ..., V_k. Each C_i is a temporal constraint (or any other type of constraint introduced below). Note that the order of constraints in a conjunction is irrelevant.

Each event `E` has (conceptually) an infinite number of *offspring* events, named `E#1`, `E#2`, etc., which are implicitly preceded by `E` itself. Events of the form `E#`N may appear in queries and in bodies, but not heads, of relation definitions. `E#`N is treated exactly like a body variable `F` such that `E«F`, except that `E#`N can be constrained by other constraints that refer to `E`. The '`#`' notation is defined in Figure 1. For example, the following query defines two sequences of events `A`, `A#1`, `A#1#1`, ..., `B`, `B#1`, `B#1#1`, ... such that `A«B`, `A#1«B#1`, etc.:

← `A «* B.`

`E «* F` ← `E « F, E#1 «* F#1.`

Execution. Logically, the solution to a Tempo query is an answer substitution giving an execution time *t* for each event *E*. In practice, Tempo computes this implicitly, by actually *executing* an event *E* at time *t*. That is, events are executed in the specified order. Events with an execution time of *eternity* are never executed. Each event is executed as soon as its predecessors have been executed, without waiting for a complete solution.

A Tempo interpreter works as follows. It *tries* each event `E` in the query, checking whether a constraint `X«E` exists (`E` is *disabled*) or not (`E` is *enabled*). Once `E` is enabled it can be *executed*: all constraints of the form `E«F` are deleted and each event `F` (previously preceded by `E`) is tried. If `E` is not disabled but appears as an argument of a user-defined temporal constraint, that constraint is *expanded* (replaced by the body of its defining clause) until `E` becomes either enabled or disabled. When a constraint is expanded, its body variables are tried. Events of the form `E#`*N* are treated in the same way as body variables, but Compilation Rule 1 is applied first, to make explicit the precedence constraints; for example, the '`«*`' definition above is translated as follows:

`E «* F` ← `E « F, E#1 «* F#1, E « E#1, F « F#1.`

Figure 2 outlines the interpreter algorithm, while Figure 3 shows a trace of the interpreter for the query `A«*B`.

Compilation Rule 1. For each event `E#`*N* appearing in a relation definition or query, add the constraint `E«E#`*N*.

2.2 Values and Functions

The language outlined above, and in (Gregory and Ramirez, 1995), cannot express computation because there is no way to represent or manipulate data. To remedy this, we now allow temporal constraints to include *data values* as well as events, distinguished by the argument positions in which they appear. A temporal relation definition may be accompanied by a declaration:

`type` $r(type_1, \ldots, type_k)$.

where each $type_i$ indicates the type of the *i*th argument of *r*: `e` (event) or `d` (data). Every temporal relation must have at least one event argument; if the type declaration is omitted, *all* arguments are events. Both arguments of the '«' primitive are events.

The types of variables in a relation definition are determined at compile time. *V* is a *data variable* if it appears in data argument positions in head and/or body constraints, or an *event variable* if it appears in event argument positions; it is not allowed in both. Event variables must appear in event argument positions of body constraints, while each data argument position must contain a *data value*, defined as a data variable, a constant, or a structured term $k(t_1,\ldots,t_j)$, where $t_1, \ldots, t_j$ are data values.

Data values can be manipulated by *function constraints*: $f(t_1,\ldots,t_j)$ where *f* is a *function relation*. *All* arguments of function constraints are data values, and are classified into input arguments and (at least one) output arguments, which must be variables. Each data variable in a relation definition must appear exactly once in a

place query constraints in constraint set CS; place query variables in try list TL;
while TL is not empty:
  delete first member E of TL;
  while E is con(C) in CS:
    replace C in CS by the body of its defining clause;
    add body variables (if any) to rear of TL;
  if E is enabled in CS then execute E.

if E appears on right of a '«' constraint in CS then E is disabled in CS
else if E appears in a user-defined temporal constraint C in CS then E is con(C) in CS
else E is enabled in CS.

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.

Figure 2: A simple Tempo interpreter

<u>CS</u>	<u>TL</u>	
{A«*B}	[B,A]	B is con(A«*B)
{A«B, A#1«*B#1, A«A#1, B«B#1}	[B,A,B#1,A#1]	B is disabled
	[A,B#1,A#1]	A is enabled: execute A
{A#1«*B#1, B«B#1}	[B#1,A#1,B]	B#1 is disabled
	[A#1,B]	A#1 is con(A#1«*B#1)
{A#1«B#1, A#1#1«*B#1#1, A#1«A#1#1, B#1«B#1#1, B«B#1}	[A#1,B,B#1#1,A#1#1]	A#1 is enabled: execute A#1
{A#1#1«*B#1#1, B#1«B#1#1, B«B#1}	[B,B#1#1,A#1#1,B#1]	B is enabled: execute B
{A#1#1«*B#1#1, B#1«B#1#1}	[B#1#1,A#1#1,B#1]	B#1#1 is disabled
	[A#1#1,B#1]	A#1#1 is con(...)
...	...	...

Figure 3: Execution of query `A«*B`

position where it is given a value (a head argument or an output argument of a function constraint); no cycles are allowed.

Although we only use familiar primitives ('+', '*', etc.) in this paper, functions can in general be defined by the user. We do not prescribe a language in which to define function relations. Any reasonable functional or logic programming language could be used for this purpose, provided that

Each function constraint can be evaluated when all of its input arguments are ground (variable free) and, after evaluation, will produce ground values for all of its output arguments.

The following example defines `clock`, representing a clock that maintains the time in hours and minutes, and increments it by one minute with each tick:

```
← clock(T, 0, 0).

type clock(e, d, d).
clock(Tick, H, M) ← +(M, 1, M1), mod(M1, 60, M2),
   //(M1, 60, H1), +(H, H1, H2), mod(H2, 24, H3),
   clock(Tick#1, H3, M2).
```

If the final argument position of a function relation f is its only output argument, we can omit the constraint $f(t_1,...,t_{j-1},V)$ and replace all occurrences of V by $f(t_1,...,t_{j-1})$. The `clock` relation can be defined using this functional syntax as follows:

```
type clock(e, d, d).
clock(Tick, H, M) ← clock(Tick#1, (H+(M+1)//60) mod 24, (M+1) mod 60).
```

Execution. All functional and logic programming languages feature something like our function constraints, but it is not obvious how these should be implemented in Tempo: the aim of a Tempo program is to execute events, not to compute values.

A simple method would be to *eagerly* evaluate function constraints as soon as they are created. Because there is no control flow in Tempo, a function that produces a value for a variable must be evaluated before its consumers. A drawback of this eager approach is that a function constraint would be evaluated even if its output is not needed. More seriously, if a function includes event values (Section 2.3), it may be impossible to evaluate it immediately. Many (concurrent or lazy) languages *suspend* functions until their input arguments are available. We rejected this option in Tempo because it would introduce implicit synchronization between the evaluation of function constraints; this would duplicate much of the functionality that Tempo provides explicitly via events, and may be complex to implement.

Our chosen strategy avoids the disadvantages of both eager and lazy evaluation. Just before expanding a temporal constraint that contains output variables of a function constraint C, we evaluate C. If the input arguments of C contain data variables output from other function constraints, those are evaluated first, and so on. This guarantees that all data arguments of a temporal constraint C are ground terms by the time that it is expanded, which in turn ensures that all function constraints created when C is expanded have ground inputs, so they can be evaluated when their outputs are required. Unlike lazy evaluation, our compiler can determine which functions need to be evaluated before expanding each temporal constraint.

2.3 Events with Values

In many cases, the relative execution time is not the only important attribute of an event. For example, an act of interprocess communication in a concurrent program can be modelled as an event that has data associated with it.

We now allow an event E to (optionally) have a data value, named E^. If an event has a data value, it must be a ground term and must be attached to the event *before* it is executed. The value can be accessed only *after* the event is executed, by which time its value (if any) is fixed. In this way, the execution of event E provides the necessary synchronization between the producer and consumer of the value E^.

An event E can be given a value t by using an assignment E^`:=`t. Assignment is a new primitive temporal relation, read logically as equality. The data value of event E can be accessed by placing E^ in a data argument of a temporal constraint, an input argument of a function constraint, or on the right of an assignment.

Figure 4 shows a concurrent algorithm for incrementally computing factorials: one process generates a stream of consecutive integers (1, 2, ...); another process receives these integers, multiplies each by the last factorial computed, and outputs the resulting factorials (1!, 2!, ...). The two processes are represented by the constraints `ints(I,1)` and `fact(I,F,1)`, and the message streams are events `I` and `F`, depicted in Figure 5 (circles represent events, numbers in them show the events' values, and an arrow from `E` to `F` shows the precedence `E«F`). The optional `F«*I#1` constraint in the query prevents integers from being generated faster than they are consumed.

```
← ints(I, 1), fact(I, F, 1), F «* I#1.

type ints(e, d).
ints(I, N) ← I^:= N, ints(I#1, N+1).

type fact(e, e, d).
fact(I, F, NF) ← F^:= NF*I^, fact(I#1, F#1, NF*I^).
```

Figure 4: Concurrent factorials program

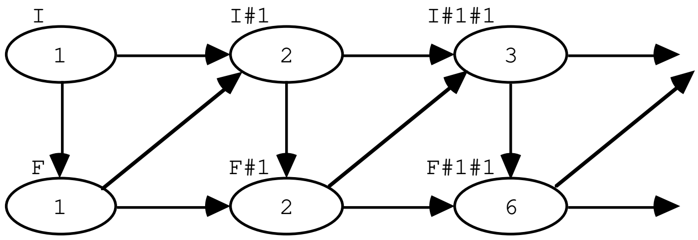

Figure 5: Events `I` and `F` in factorials program

Execution. The implementation must ensure that (a) values are given to events before they are executed and (b) accessed after they are executed.

The first requirement can be satisfied by performing an assignment E`^:=`V before E is executed. However, it cannot be done arbitrarily early, in case the value of V is not available. Our solution is to do the assignment *before* executing E, but *after* E is enabled. The second requirement is to ensure that the value D^ is not accessed until D has been executed. Since D^ may appear either in a user-defined temporal constraint or on the right of an assignment, there are two cases to consider:

1. An assignment E`^:=`t must be delayed until the events on which t depend have been executed, to avoid assigning a non-ground value to an event. This is achieved by Compilation Rule 2, which exploits the fact that an assignment to E is not performed while E is disabled. For example, in the `fact` clause of Figure 4, the constraint `I«F` is automatically added, delaying the assignment `F^:=NF*I^` until `I` is executed and therefore has a value.

   Definition. Term t *depends on* event D if t is D`^`, or t is a structured term containing a value that depends on D, or t is an output variable of a function constraint with an input argument that depends on D.

   Compilation Rule 2. For each assignment E`^:=`t in a conjunction Cs, and for each event D on which t depends, add the constraint D«E to Cs .

2. A user-defined temporal constraint C must not be expanded until all of its data arguments are ground, i.e., until the events on which they depend have been executed. This is achieved by Compilation Rule 3, which exploits the fact that C will not be expanded while all of its event arguments are disabled. For example, in the `fact` clause of Figure 4, the constraints `I«I#1` and `I«F#1` are added, delaying the expansion of `fact(I#1,F#1,NF*I^)` until `I` is executed.

   Compilation Rule 3. For each user-defined temporal constraint C in a conjunction Cs, and for each event D in $DATA_C$ and E in $EVENT_C$, add the constraint D«E to Cs. $DATA_C$ is the set of events on which the data arguments of C depend; $EVENT_C$ is the set of C's event arguments.

2.4 Disjunction

As well as atomic constraints, a constraint may be a disjunction (Cs_1`;` ...`;` Cs_n), where each Cs_i is a conjunction of constraints (called an *alternative*, to distinguish it from the *top-level conjunction*). The order of alternatives in a disjunction is not significant. Constraints inside disjunctions must be atomic (disjunctions cannot be nested) and, to avoid problems with variable scope, cannot be function constraints. Disjunction has two roles in Tempo: to test data values and to allow non-determinism in the order of events, which is essential in specifying concurrent phenomena. We describe each of these in the next two subsections.

Conditionals

Disjunction can be used to test data values, like conditionals in conventional languages, using a new kind of constraint: a *test constraint* $t(t_1,...,t_j)$ where t is a *test relation*. Like a function constraint, all arguments are data values, but a test constraint has no output arguments and may fail. We only use familiar primitives ('==', '<', '>=', etc.) in this paper, but test relations may also be defined by the user in any suitable functional or logic programming language.

For example, suppose that we wish to generate all factorials up to (say) 10000. The infinite generator of Figure 4 could be changed into a finite program by redefining the `ints` and `fact` relations with an additional "counter" argument. However, a much more elegant method is possible: simply add to the query an extra constraint `upto(10000,F)`, constraining all descendants of the first descendant of `F` whose data value is greater than 10000 to be eternity events. The following query will terminate after generating the first factorial greater than this limit:

```
← ints(I, 1), fact(I, F, 1), F «* I#1, upto(10000, F).

type upto(d,e).
upto(X, F) ←  X < F^, F#1 « F#1;
              X >= F^, upto(X, F#1).
```

Execution. If any test constraint fails, the alternative containing it is deleted from the disjunction. When the disjunction alternatives contain mutually exclusive tests, as in the examples above, the disjunction becomes "flat" and is handled in the normal way. The main implementation issue is to ensure that tests are not evaluated before their arguments are ground (i.e., all events on which their arguments depend have been executed). This is achieved by the following:

> ***Compilation Rule 4***. For each disjunction J, add (at the top level) constraints T«U for each T in $TESTED_J$ and U in $USED_J$. $USED_J$ is the set of events that appear in J on the right of '«' constraints or in user-defined temporal constraints; $TESTED_J$ is the set of events on which arguments of tests in J depend.
>
> ***Restriction 1***. For each disjunction J, $USED_J$ and $TESTED_J$ must be disjoint.
>
> ***Disjunction Rule***. While checking whether event E is enabled, a disjunction J such that E is in $USED_J$ is tried only if E is not disabled at the top level; i.e., disjunctions are tried after '«' constraints and user-defined temporal constraints.

For example, for the disjunction in the `upto` definition, *TESTED* is {`F`} and *USED* is {`F#1`}, so the compiler adds the (redundant) constraint `F«F#1`.

Non-determinism

Disjunctions do not necessarily include test constraints and, even if they do, they need not be mutually exclusive. Then the role of the disjunction is to express non-determinism in the order of events. For example, Figure 6 defines a stack: the temporal constraint `stack(Push,Pop)` describes the relation between a sequence of *push* events and a sequence of *pop* events; the data value of each event is the value to be *push*ed or *pop*ped. The disjunction reflects non-determinism: at any time, if the stack is non-empty (`list(L)`), the next transaction may be either a *push* or a *pop*.

```
← producer(Push), stack(Push, Pop), consumer(Pop).

stack(Push, Pop) ← stack1(Push, Pop, []).

type stack1(e, e, d).
stack1(Push, Pop, L) ←
   Pop«Push, list(L), Pop^:=head(L), stack1(Push, Pop#1, tail(L));
   Push«Pop, stack1(Push#1, Pop, [Push^|L]).
```

Figure 6: Stack

Execution. If the alternatives of a disjunction contain mutually inconsistent temporal constraints, as in Figure 6 (`Pop«Push` and `Push«Pop`), the interpreter must select an alternative to use. The way in which this choice is made is a key aspect of the design of Tempo.

While checking whether an event E is enabled and the constraint set contains a disjunction J such that E is in $USED_J$, all tests in J are first evaluated (as explained above). If the result is still a disjunction, each alternative is checked: If E is enabled in at least one alternative but not all (E is *conditionally enabled*), J is *reduced* by deleting all alternatives in which E is disabled. Since disjunctions are tried only if E is not disabled at the top level (by the Disjunction Rule), E can now be enabled and progress can be made. The programmer can control when a disjunction is reduced by using sufficient precedence constraints at the top level. For example, in the query of Figure 6, `Push` will not be enabled by the `stack(Push,Pop)` constraint until `producer(Push)` unconditionally enables it.

To simplify implementation, temporal constraints can be expanded, and assignments done, only at the top level. For this reason the programmer must observe the following:

Restriction 2. If event E appears in a disjunction in a user-defined temporal constraint, but is not disabled in the same alternative, E must be disabled at the top level.

Restriction 3. If event E is enabled in a disjunction alternative that has an assignment to E, but E is not disabled in all other alternatives, E must be disabled at the top level.

2.5 Concurrency

The above language can specify concurrent systems, and these specifications can be executed by our sequential interpreter. However, as a concurrent programming language, Tempo also allows a query to be *solved* concurrently. If any group of constraints in a conjunction is enclosed in braces (not affecting the logical reading), it will be executed as a separate process. For example, a query may take the form

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

Each process solves its own constraints independently, except where it shares events with other processes; a shared event is executed when it has been enabled by *all* processes that share it. Shared events provide the communication medium between Tempo processes.

Execution. A constraint of the form `{`Cs_i`}` spawns a new invocation of the Tempo interpreter to solve Cs_i. Each interpreter works in the way that we have described, except that an event cannot be executed as soon as it is found to be enabled. Instead, when a process *P* finds that event *E* is enabled by the constraints in *P*, it informs all other processes that share event *E*, and passes them the value (if any) that *P* has given to *E*. When all processes have done this, *E*'s value is communicated to all of them, which then treat the event as executed.

The Disjunction Rule of Section 2.4 is hard to implement in a concurrent interpreter because no single process has access to all constraints. We therefore modify it so that a process must wait for all other processes to enable an event before checking it against a disjunction:

> ***Disjunction Rule'***. While checking whether event *E* is enabled, a disjunction *J* such that *E* is in $USED_J$ is tried only if *E* is not disabled at the top level *and E* has been (unconditionally) enabled by all other processes.

> ***Restriction 5***. Two disjunctions *J* and *K* must appear in the same process unless $USED_J$ and $USED_K$ are disjoint.

Also, Compilation Rules 1–4 must be modified so that their explicit '«' constraints are added to *every* process where they are needed.

For example, the stack query of Figure 6 can be executed as three processes: two clients, sending "messages" on `Push` and `Pop`, and a server that receives messages from each client:

```
← {producer(Push)}, {stack(Push, Pop)}, {consumer(Pop)}.
```

The server process `{stack(Push,Pop)}` initially waits for either `Push` or `Pop` to be enabled by its respective sender, because of the new Disjunction Rule. If `Push` is then enabled by the `{producer(Push)}` process, the server will conditionally enable `Push` and commit to the second alternative of its disjunction. Alternatively, if `{consumer(Pop)}` enables `Pop`, the server will test that `L` is non-empty and, if so, will conditionally enable `Pop` and commit to the first alternative of its disjunction.

3 Programming in Tempo

3.1 Example: Bounded Buffer

In Figure 4, the `F«*I#1` constraint has the effect of placing a buffer of size 1 on the `I` channel. This can be generalized to a bounded buffer of any specified size *N* that can be placed between any producer and consumer process, as shown in Figure 7. A different way to implement a buffer of size *N* is as a chain of *N* processes, each containing a buffer of size 1; Figure 8 shows this algorithm written in Tempo.

To show that these two algorithms are equivalent (in respect of their safety properties), it is only necessary to prove that

`bounded_buffer(R, S, `*N*`)` $\leftrightarrow$ `bounded_buffer_m(R, S, `*N*`)` and this can be done quite simply using standard logic program transformation techniques. Space limitations prevent the inclusion of the proof here, but similar proofs were presented in (Gregory, 1996), in the original subset of Tempo.

```
← {producer(R)}, {bounded_buffer(R, S, N)}, {consumer(S)}.

type bounded_buffer(e, e, d).
bounded_buffer(R, S, N) ← buffer(R, S), bounded(R, S, N).

buffer(R, S) ← S^:=R^, buffer(R#1, S#1).

type bounded(e, e, d).
bounded(R, S, N) ← N >= 2, bounded(R#1, S, N-1);
                   N == 1, S «* R#1.
```

Figure 7: Single-process bounded buffer program

```
← {producer(R)}, {bounded_buffer_m (R, S, N)}, {consumer(S)}.

type bounded_buffer_m(e, e, d).
bounded_buffer_m(R, S, N) ←
   N >= 2, buffer1(R, M), bounded_buffer_m(M, S, N-1);
   N == 1, buffer1(R, S).

buffer1(R, S) ← {buffer(R, S), S «* R#1}.          % Spawn buffer process
```

Figure 8: Multi-process bounded buffer program

3.2 Servers

Many-to-One Communication. A *server* is a process that provides a service to a variable number of *client* processes. To implement servers in Tempo we use a technique similar to that used in concurrent logic programming languages: message streams (linear event structures) from all clients are combined, using a `merge` constraint, into a single stream which is passed to the server. For example, the stack program of Figure 6 can be rewritten for two producers as follows (this can easily be extended to allow a larger, dynamically varying, number of clients):

```
← {producer(Push1)}, {producer(Push2)}, {consumer(Pop)},
   {merge(Push1,Push2,Push), stack(Push, Pop)}.

merge(M1, M2, M) ← M1 « M2, M^:=M1^, merge(M1#1, M2, M#1);
                   M2 « M1, M^:=M2^, merge(M1, M2#1, M#1).
```

Bidirectional Communication. Servers usually return a response to requests received from clients. One way to arrange this in Tempo is by providing one event structure to carry requests from client to server and another to convey replies back to the client; whenever a new client is created, a `merge` constraint must be introduced in the request stream and a `split` constraint in the reply stream. The problem with this technique is that it works only if replies are generated by the server in the same order as requests are received. This is not generally true: for example, an FTP server may answer a request for a small file before sending a larger file requested earlier.

A better way to handle bidirectional communication in Tempo is by using *request/reply structures*: binary event trees of the form shown in Figure 9. Here, the value of event `E` is a request, the value of `E#2` is the reply, while `E#1` is (as usual) the next request in the sequence. Each request event precedes both the corresponding reply and the next request, but the replies may be sent in any order. These structures can be combined almost as easily as linear event structures. Instead of `merge`, we use the `merge2` relation, defined as follows:

```
merge2(M1, M2, M) ← merge(M1, M2, M), mergeb(M1, M2, M).

mergeb(M1, M2, M) ← M1«M2, M1#2^:=M#2^, mergeb(M1#1, M2, M#1);
                    M2«M1, M2#2^:=M#2^, mergeb(M1, M2#1, M#1).
```

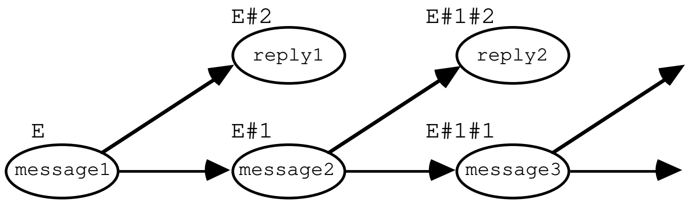

Figure 9: Request/reply event structures

Example: Disk Scheduler

Consider an operating system in which user processes perform disk I/O operations by sending requests to a disk driver process. This can be represented by a Tempo query

```
← {clients(D)}, {disk(D)}.
```

where `D` is a request/reply structure with `D`, `D#1`, etc. labelled by disk I/O operations (*read* or *write* from/to a given disk address); `D#2`, `D#1#2`, etc. are labelled with the replies. `clients(D)` defines the combined output `D` (produced by `merge2`) of all clients. `disk(D)` is the disk driver, which replies to I/O requests in order of arrival.

A disk scheduler tries to minimize disk head movement by rearranging pending I/O requests by cylinder number before passing them to the disk driver. This is expressed in Tempo as a constraint `scheduler(C,D)` between two request/reply structures:

```
← {clients(C)}, {scheduler(C, D), disk(D)}.
```

Figure 10 shows a definition of `scheduler(C,D)` which implements the C-SCAN scheduling algorithm (Andrews, 1991). It has a state comprising a list, initially empty, of pending I/O requests and the current cylinder position `Pos` of the disk head. When an request is received on `C` (`C«D`), it is stored in the pending requests list together with a unique identifier. A `match` constraint searches for the request with a matching identifier among `D` and its descendants, and copies its reply to `C#2`. When the disk is ready (`D«C`) and there are pending requests (`list(Buff)`), the best of them, according to the current disk position, is removed from the buffer and passed to the disk driver. The function `select(Pos,Buff,Best,Rest)`, not defined here, searches `Buff` for the entry `Best` with the smallest cylinder number no less than `Pos`, if one exists, else the smallest cylinder number of all; `Rest` is the remainder of `Buff`.

4 Conclusions

4.1 Summary

Tempo is declarative in that the order of events, and their values, is determined only by the constraints in a program, *not* by (e.g.) the order of items in conjunctions and disjunctions, the use of braces, or the difference between '`:=`' and '`==`'. This makes it

```
scheduler(C, D) ← sched(C, D, T, [], 0, 1), D «* T.

type sched(e, e, e, d, d, d).
sched(C, D, T, Buff, Pos, Id) ←
   select(Pos, Buff, Best, Rest),
   (  C « D,
         match(C#2, D, T, Id),
         sched(C#1, D, T, [(C^, Id) | Buff], Pos, Id+1);
      D « C, list(Buff),
         D^:= arg(1, Best), T^:= arg(2, Best),
         sched(C, D#1, T#1, Rest, D^, Id)   ).

type match(e, e, e, d).
match(Cr, D, T, Id) ←  T^ == Id, Cr^:= D#2^ ;
                       T^ \== Id, match(Cr, D#1, T#1, Id).
```

Figure 10: C-SCAN disk scheduler

easy to develop concurrent algorithms: the constraints are defined first and then they can be grouped into parallel components using braces. Standard techniques can be used to prove programs equivalent and to derive programs from specifications, all expressed in Tempo (Gregory, 1996). (It is not possible to prove progress properties — that a given event will eventually be executed — without recourse to the operational semantics. For this, the language provides an implicit fairness guarantee: every unconditionally enabled event will eventually be executed.)

Tempo embodies a novel, unconventional programming paradigm: *events* are first-class objects in the language while *state* is implicit, being represented by data arguments of constraints. A major advantage of this is that it makes possible Tempo's constraint-oriented style of specifying events: the execution time, and the number of offsprings, of *E* are described by the conjunction of a number of constraints on *E*. This in turn allows programs to be composed by conjunction (as in the `upto` example of Section 2.4 and the `merge2` example of Section 3.2).

A prototype implementation of Tempo has been used to test all examples in this paper; it is available from `http://star.cs.bris.ac.uk/software/tempo.html`. Although high-performance implementations have not yet been developed, we believe that Tempo can be implemented efficiently. Computation, performed by function constraints on data variables, is separated from synchronization and communication, which involves events, so Tempo can be considered a coordination language, which adds concurrency to any computation language, declarative or otherwise. Ramirez (1996) proposes a language, Tempo++, that combines Tempo (for synchronization) with a computation language such as Prolog in an object-oriented framework.

4.2 Comparison

Tempo as a Specification Language. As a specification language for concurrent systems, Tempo is closer to languages based on temporal logic, e.g., Unity (Chandy and Misra, 1988; Misra, 1995) and TLA (Lamport, 1994), than process algebras (Hoare, 1985; Milner, 1989). Unity and TLA specifications can express progress properties of algorithms, as well as safety properties, but are not executable. Like Tempo, TLA uses classical logic as far as possible, restricting the use of temporal logic to specify progress. In Tempo, as in Unity and TLA, a specification of an algorithm's properties can be refined to a lower-level form that represents the algorithm itself: an algorithm is correct if it logically implies the specification (Gregory, 1996). Unity and TLA both model concurrent systems by sequences of actions that modify a single shared state. This is adequate for specifying and reasoning about systems, but the shared state might be a bottleneck if the specifications were to be executed. In contrast, Tempo's event sequences are independent unless explicitly constrained; this allows a specification of a distributed system to be executed in an equally distributed manner.

Tempo as a Programming Language. We have already mentioned how Tempo replaces the extralogical control features (e.g., modes and sequential operators) of concurrent logic programming by declarative equivalents. From a programming viewpoint, the main difference between the two types of language is the use of the logical variable. In concurrent logic programming languages, a variable can be included in a data structure and, at the same time, act as a communication channel. This allows for some elegant programming techniques; for example, the disk scheduler of Section 3.2 can be implemented elegantly by including reply variables in request messages. On the other hand, implementation can be expensive because every variable is potentially shared, so suspension, locking, etc., are necessary. In contrast, Tempo follows a principle advocated by the designers of Linda (Carriero and Gelernter, 1989): to separate computation from coordination. Tempo events can have data values, but events cannot appear in data structures.

Logical variables effectively provide a global address space: any two processes that receive a message containing a variable can use it to communicate directly. Global communication, of a very different form, is also provided by the Linda model. Tempo has more in common with message passing languages: processes can communicate directly only if they share events. This is much more efficient to implement in a distributed system.

Concurrent constraint programming (Saraswat, 1993) generalizes concurrent logic programming to allow the values of variables to be defined by a conjunction of constraints imposed incrementally and concurrently. Another family of declarative languages is based on executable temporal logic (Moszkowski, 1986; Abadi and Manna, 1987; Fisher, 1993; Cleary and Smith, 1996). Although, in principle, temporal logic can express safety and progress properties of concurrent programs, we are not aware of a concurrent programming language entirely based on temporal logic.

4.3 Future Work

We are currently introducing the concept of real time into Tempo, to allow real-time applications to be tackled. Real-time extensions have been made to many concurrent languages, including Parlog and concurrent constraint languages (Saraswat *et al.*, 1994; De Boer and Gabbrielli, 1995), but it is a particularly natural development of Tempo, since the execution time of events is already a first-class concept in the language. Although Tempo provides a novel constraint-oriented way to specify events, their *data values* are simply assigned; these too might usefully be defined by constraints, along the lines of existing constraint programming languages (Van Hentenryck, 1989; Saraswat, 1993). Several other extensions could be made to the language: event offsprings could be given meaningful names (e.g., `M#reply` instead of `M#2`); events might have duration instead of being atomic; and so on.

As well as the language design, we plan to apply Tempo to large-scale concurrent applications, develop an efficient implementation of Tempo (probably by building on an existing declarative language implementation and using that language to define

Tempo's function and test constraints), and to make detailed comparisons with other concurrent programming and specification languages.

References

Abadi, M. and Manna, Z. 1987. Temporal logic programming. *Proc. of the Symp. on Logic Programming*, S. Haridi (Ed.). IEEE Computer Society Press, pp. 4–16.

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

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

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

Cleary, J.G. and Smith, D.A. 1996. Declarative I/O, assignment and concurrency. Technical report. Dept. of Computer Science, Univ. of Waikato.

De Boer, F.S. and Gabbrielli, M. 1995. Modelling real-time in concurrent constraint programming. In *Proc. of the Intl. Logic Programming Symp.*, J.W. Lloyd (Ed.). MIT Press, pp. 528–542.

Fisher, M. 1993. Concurrent Metatem: a language for modelling reactive systems. *Proc. of Parallel Architectures and Languages Europe*.

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

Gregory, S. 1996. Derivation of concurrent algorithms in Tempo. In *Proc. of the 5th Intl. Workshop on Logic Program Synthesis and Transformation*, M. Proietti (Ed.). Springer-Verlag, pp. 46–60.

Gregory, S. and Ramirez, R. 1995. Tempo: a declarative concurrent programming language. In *Proc. of the 12th Intl. Logic Programming Conf.*, L. Sterling (Ed.). MIT Press, pp. 515–529.

Hoare, C.A.R. 1985. *Communicating Sequential Processes*. Prentice Hall.

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

Lamport, L. 1994. The temporal logic of actions. *ACM Trans. on Programming Languages and Systems 16*, 3, pp. 872–923.

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

Misra, J. 1995. A logic for concurrent programming. *J. of Computer and Software Engineering 3*, 2, pp. 239–300.

Moszkowski, B. 1986. *Executing Temporal Logic Programs*. Cambridge Univ. Press.

Pratt, V. 1986. Modeling concurrency with partial orders. *Intl. J. of Parallel Programming 15*, 1, pp. 33–71.

Ramirez, R. 1996. *A Logic-Based Concurrent Object-Oriented Programming Language*. PhD thesis. Univ. of Bristol.

Saraswat, V.A. 1993. *Concurrent Constraint Programming*. MIT Press.

Saraswat, V.A., Jagadeesan, R., and Gupta, V. 1994. Foundations of timed concurrent constraint programming. In *Proc. of the 9th IEEE Symp. on Logic in Computer Science*, S. Abramsky (Ed.). IEEE Computer Society Press.

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

Van Hentenryck, P. 1989. *Constraint Satisfaction in Logic Programming*. MIT Press.