

Computing Time Intervals Logically

André Vellino
William Older
Bell-Northern Research
Ottawa, Ontario
e-mail:vellino,older@bnr.ca

Abstract

The purpose of using first-order logic to program a computer is to turn the task of programming into the task of expressing relations that must be jointly satisfiable. One obstacle to relational and logical programming has been arithmetic, which is usually evaluated functionally. This paper describes a method for expressing arithmetic relations declaratively which computes on intervals in the real line. This technique allows a formulation of an interval-based temporal logic that is directly executable as a program.

1 Introduction

Programming computers in the language of first order logic has been going on now for over 20 years. The language of Prolog (*Programming in logic*) was invented by Alain Colmerauer in the early 70s primarily as a means for expressing rules of grammar for natural language translation. Since then, the language of logic has been successfully applied in just about every situation where computers can be used (from scheduling airplanes at airports to controlling the processing of telephone calls for a voice-messaging system).

One of the principle motivations for exploring logic as a programming language was the hope that the task of programming would become increasingly similar to the the task of specifying *what* software should do rather than *how* it should execute. This effort to move away from imperative programming

towards declarative programming has met with many successes but there are some computing tasks that are particularly ill-suited to a declarative relational formulation. For example, operations that irrevocably change the state of the world (such as writing output to a file) seem to require imperative action statements rather than declarative relational statements.

One obstacle to achieving the end of truly declarative, relational programming in logic has been arithmetic. In most programming languages arithmetic operations are performed by a functional evaluation mechanism. Until recently this was also true for logic programming languages like Prolog. Arithmetic in Prolog is essentially no different from arithmetic in Fortran or 'C'. The functional nature of Prolog arithmetic implies that arithmetic expressions will not, in general, commute with one another. For example:

Y is 2 * Z, X is Y + B

is not, in general, equivalent to

X is Y + B, Y is 2 * Z

But since “,” (comma) is interpreted in Prolog as the logical “and” relation, this non-commutativity of arithmetic operations is a major weakness in the logical structure of the Prolog language.

Such weaknesses have been addressed by constraint logic programming systems such as the one described in [7] and [5]. In this paper we want to describe another technique for doing arithmetic logically that was first expounded by John Cleary [6], but whose fundamental principles can be found in [8] and [9]. Several implementations of Prolog have incorporated these ideas [3] [12] and have been applied to practical problems [10][4]. One particularly interesting application of this system is to express and compute temporal relations on time intervals. However, the method of constraint interval arithmetic for doing this kind of temporal logic differs from the conventional AI temporal theorem provers [1] in that it “reasons” with time intervals on a quantitative time-line rather than *about* the temporal relations.

2 Relational Interval Arithmetic

We begin by providing an introduction to the kinds of relations that can be formulated in a system that can express arithmetic relations on intervals in

the real line. The first notion this language needs is the idea of a variable constrained to lie within bounds (real, integer or boolean). We denote this by writing

$$X : \text{real}(Lx, Ux) \quad I : \text{integer}(Li, Ui) \quad B : \text{boolean}$$

where X I and B are logic variables which become constrained to lie between the floating point bounds $[Lx, Ux]$ or the integer bounds $[Li, Ui]$ or the boolean bounds $[0, 1]$. An integer interval is a real interval whose bounds are integral and a boolean is represented by an integer interval between 0 and 1.

Constraints on variables ranging over reals can express equalities ($==$) and inequalities ($=<$ and $>=$) of expressions. The arithmetic expressions themselves can be any well formed arithmetic formula composed from $\{*, +, \div, -, \odot\}$ where \odot is any one of the transcendental functions (\sin, \cos, \tan) or their inverses. Constraints on integers include disequalities ($<>$) and boolean expressions can be constrained by the relations $\{\text{or}, \text{and}, \text{not}, \text{xor}, \text{nand}, \text{nor}\}$. In addition there are the constraints that force two intervals to have the same upper (lower) bounds that ensure containment. As we will see in the second half of this paper these constraints can be used to express temporal relations.

A typical query to such a system might be:

$$?- X:\text{real}(3,7), Y:\text{real}(2,8), Z:\text{real}(12,25), X*(X+Y)==Z.$$

whose answer narrows the intervals (relative to the question) to:

$$\begin{aligned} Z &: \text{real}(15.0, 25.0), \\ X &: \text{real}(3.0, 5.0), \\ Y &: \text{real}(2.0, 5.333333333333333). \end{aligned}$$

In general terms, relational interval arithmetic can be described as a method for taking a set of mathematical relations between certain quantities, and constructing from them a process which maps initial intervals on all quantities to final intervals for these quantities. It does this by constructing a proof (using interval fixed point iteration) that any solutions in the initial intervals must lie in the final intervals. The mapping from initial to final intervals is contracting (the final intervals are subintervals of their initial values), idempotent (repeating the same constraint has no effect on the interval values), and inclusion monotone (smaller initial intervals yield smaller final

intervals). Moreover, the mapping from initial to final intervals is independent of the order in which constraints are imposed and of the details of the implementation.

There are two ways one can quantify over a variable that refers to an interval. Under the *existential* interpretation, a real (integer) interval represents an unknown real (integer) number that lies somewhere between the upper and lower bounds. Under the *universal* interpretation, an interval is the set of all real (integer) numbers between its upper and lower bounds. (This is analogous to the two possible interpretations of a logic variable in standard logic programming: one where the variable refers to something specific (but unknown) and the other where the variable refers to the set of all its valid instantiations.) In the system that we are considering here, the existential interpretation is (most of the time) the correct one, although it is often appropriate to use the universal one when dealing with temporal relations.

2.1 Constraint Propagation

So what happens when an arithmetic expression containing intervals is evaluated? First it is transformed into a sequence of primitive interval operations which is then used to construct a constraint network similar to a data flow network. Data may flow (as a wave of “narrowing”) through the network in any direction, but always from the more constrained to the less constrained variables. The undirected nature of the graph corresponds directly to its ability to represent relations and not just functions. The flow of data may be in a cyclic pattern corresponding to a fixpoint iteration, just as may occur in a spreadsheet system. At any time during the constraint propagation an inconsistency may be detected. As described above, this terminates the iteration and reports failure to the Prolog engine.

The primitive elements of the constraint network satisfy three properties. First, interval variables can only narrow the ranges of their input intervals, i.e. intervals can never grow. Secondly, all the primitive constraint equations or inequalities should be idempotent operations, i.e. applying an operation twice is the same as applying it once. Third, these operations on intervals should be isotone, i.e. preserve set inclusion order. More precisely, if an interval which has an initial range R_i narrows to a final range R_f , then, any other initial range $Q_i \subseteq R_i$ will narrow to a final range Q_f such that

$Q_f \subseteq R_f$.

When these three properties are satisfied by the individual primitive operations, arbitrary sequences of such operations will converge to results that are independent of the actual order of operations. By lattice-theoretic arguments, it has been shown [11] that the total operation will also be narrowing, idempotent, and isotone and that successive macro operations commute and satisfy an associative law.

Two techniques are used to ensure that the truth of the algebraic laws of the real numbers is preserved. One is the use of outward rounding in all computations—the fundamental principle of functional interval arithmetic introduced in [8]—which ensures that all computed intervals are large enough to contain all “true” answers. The other is the use of interval intersection to implement equality. The idempotent, commutative, and associative laws of intersection correspond to the reflexive, symmetric, and transitive laws of equality, respectively.

This approach has the effect that all detected inconsistencies have the force of a constructive proof that no solution exists (over the mathematical reals) for the stated problem. However, if no inconsistency is detected, the computed narrowings must be given a conditional interpretation: if solutions exist given the initial interval, they must be contained in the output intervals.

2.2 Computations as Proofs

From the point of view of logic, one of the interesting aspects of constraint arithmetic on intervals, is that its computations are *proofs*, and these are always proofs of the non-existence of solutions. As proofs, they carry a degree of logical force generally absent from traditional numerical computing, which is concerned only with heuristic methods. The fact that its proofs are always non-existence proofs makes it very different from traditional exact (rational) arithmetic, in which computations can be thought of as constructive proofs of existence.¹

¹It is precisely because interval proofs are non-existence proofs that they can be interpreted as referring to the mathematical reals, even though only bounded precision constants actually appear in the proofs themselves and the constraint system itself has no notion of “real number” in the full mathematical sense. (Of course, these proofs refer as well to the rationals, computable reals, and non-standard reals.)

3 Temporal Logic

In this section we present the essential features of a system for expressing temporal logic constraints in the language of interval arithmetic. It should be noted at the outset that this approach differs from the time representation schemes that have been used in AI, particularly those used in planning systems by various authors such as Allen and Hayes in [2]. The latter, in particular, treats intervals as primary and introduces a full set of relations between intervals (during, before, after, meets, equal,...). Most of these “second order” relations cannot be expressed in BNR Prolog interval arithmetic, or can be expressed only indirectly, but many of the inferential complexities are thereby reduced, and the ability to deal with time as a metric quantity is added.

Much of the technical work in this area has focussed on the problems of efficient inferencing over temporal relations, using inference engines specialized for whatever system of temporal relations is being proposed. When the sequencing information is given abstractly purely in terms of partial orders of events, the inferencing required involves (rather complex) transitive closures. If time is treated as metric as well, then additional complications arise to relate numerical times to time orders. In the formalism we describe here, most of these concerns are relegated to the interval arithmetic subsystem and need not belabored further, although interval arithmetic is not necessarily the most efficient mechanism for doing specific kinds of such temporal reasoning.

3.1 Interval Temporal Relations

We begin by observing that intervals can be constrained by various operations that are useful for expressing temporal relations. In particular:

- $X \leq Y$ constrains X to be inside Y
- $X \text{ -} < Y$ constrains X and Y to have the same lower bound
- $X \text{ > -} Y$ constrains X and Y to have the same upper bound
- $X == Y$ constrains X and Y to have the same bounds
- $X = < Y$ constrains $\text{ub}(X) = < \text{ub}(Y)$ and $\text{lb}(X) = < \text{lb}(Y)$

Each of these relations is transitive and has a converse. They are used to express temporal relations in the following way.

3.1.1 Equality

The interval equality relation $X==Y$ constrains the intervals X and Y to have the same start and finish points. Its explicit use is seldom required but it often arises indirectly as a consequence of the other relations. Its negation², $\text{not}(X==Y)$, succeeds only if X and Y have no overlap.

Note that equality is related to all the other relations by:

$$\begin{aligned} X==Y & \quad \leftrightarrow \quad X \text{ -< } Y \ \& \ X \text{ >- } Y \\ X==Y & \quad \leftrightarrow \quad X \text{ =< } Y \ \& \ X \text{ >= } Y \\ X==Y & \quad \leftrightarrow \quad X \text{ <= } Y \ \& \ X \text{ => } Y \end{aligned}$$

3.1.2 Before/After

The $=<$ (and $=>$) corresponds to a weak notion of before (after): $X =< Y$ constrains the interval X to be before interval Y in the weak sense that X starts before Y starts and X finishes before Y finishes, but allows the two intervals to overlap. The strong sense of before (“serial before”) in which X finishes before Y starts cannot be imposed directly as a constraint but can be *tested* by using $\text{not}(X>=Y)$. (The inability to constrain a serial relation is due to the inability to constrain non-overlap, which is due to the need to enforce an arbitrary point of separation.) The formulation of overlap constraints is described below.

3.1.3 Inclusion

The inclusion relation $X <= Y$ constrains X to be a subinterval of Y . This can be used to handle the causal dependencies if B is a component of A and causally dependent on A , this may be expressed using conditions:

$$\begin{aligned} & \text{exists}(A, Ta), \\ & \text{exists}(B, Tb), \\ & Tb <= Ta \end{aligned}$$

²Here we mean negation by failure, as with conventional Prolog. This issue is discussed further in section 3.3.

If one similarly expresses the dependency of C on B, then the transitivity of the subinterval relation ensures that $T_c \leq T_a$. If C is an essential part of A, so that $T_a \leq T_c$ as well, then all three intervals will be constrained to be coterminous (equal).

3.1.4 Overlap

We can define the temporal overlap relation between two intervals X and Y (with $X \leq Y$) by introducing a new interval T (to represent the overlap) and the condition that $Y \text{--} < T$ and $T \text{--} > X$. If X and Y ever narrow such that they no longer overlap, then T becomes empty and forces failure.

3.2 Histories and Events

All these relations deal with order relations among intervals and can be used to express complex temporal relations between such intervals. These are sufficient to express constraints on the sequence of events in a system. Specifically, we first consider how to represent a fixed, closed history of events. Then the model is modified slightly so that it can construct a completed history given an initial partial history. Finally, it is extended to deal with open histories, i.e. unfinished histories, where some of the event are still in the future.

The basic construct of this model is a notion of a history as a collection of facts, each with an associated time interval during which it is true (or known to be true). The facts are expressed as in prolog; the time intervals are notionally expressed as closed intervals ³.

The notation which we will use for temporal facts has the general form:

$$F(\text{Args}..) @ T$$

where T denotes the time interval over which F(Args..) is true. For example,

$$\text{in}(\text{bathroom}, \text{with}(\text{knife}, \text{colonel})) @ [930, 1000]$$

Primitive queries against the history are expressed in the same notation. If the bounds on the time interval are unknown the query is over all time; each successive answer will bind T to the appropriate interval of validity.

³Intervals on the reals in BNR Prolog are always closed intervals.

If T is initially bound to some definite interval, the query will return only answers falling within that interval. The use of intervals (as usual) allows for different interpretations. The simple first order interpretation is that T stands for a specific time; if its value is an interval it merely indicates that any time within that interval is useable. The second order interpretation makes the interval notion more explicit; in this case an expression such as that above would correspond (roughly) to a first order sentence like:

$$\exists t_1 \exists t_2 ((t_1 < t_2) \wedge \forall t ((t_1 \leq t) \wedge (t \leq t_2) \rightarrow F(Args..)))$$

Such sentences are quite cumbersome to deal with routinely, so the use of intervals represents a very intuitive and compact way of expressing the idea of a condition holding over a span of time. Note also that this idea also cannot be expressed so easily using any of the standard modal operators usually employed in temporal logic (such as always, never, henceforth, and so on). It is also important to note that Prolog works by instantiation, which is generally stronger than existential quantification, since knowing that something exists is not as useful as knowing what it is that exists. This is particularly important in this case, since from two sentences of the above form no conclusion can be drawn, while if the interval bounds are known explicitly a conclusion can be drawn.

The simplest conjunctive queries have the form:

$$[P @ T, Q @ T]$$

which expresses the idea that both P and Q must be true at the same time (or over the same interval).

Queries can also involve more than one time. For example, to express the idea that condition P occurred between an occurrence of Q and one of R one can use a query like:

$$[Q @ T1, T1=<T2, P @ T2, T2=<T3, R @ T3].$$

Thus the inequality operations can be used to express a notion of “before” , “after” and “between”. Note in particular that the order of conjuncts in such expressions is logically irrelevant, (although there may be operational reasons to prefer one order to another). Somewhat more complex conditions can be expressed by allowing for arithmetic on time intervals, although this requires a general assumption that time is a metric and not just an order

scale. Thus the idea of “T2 is at least N units later than T1” can be written as the constraint $T2 - T1 \geq N$, and so forth. Defined predicates are written as one would expect , e.g.

$$R @ T :- P @ T, Q @ T .$$

defines a new predicate $R @ T$ to mean the conjunction of P and Q. Note that this authorizes the inference that R is true whenever both P and Q are true; thus P and Q are sufficient conditions for R. Both P and Q will also be necessary conditions for R provided there are no other clauses authorizing the inference of R.

A natural extension of this notation is to allow several conclusions from the same set of antecedents, such as:

$$[R,S] @ T :- P @ T, Q @ T .$$

As a concrete example, let us suppose the following history is given for two switches that control a light:

```
switch1(on) @ [1,2].
switch1(off) @ [2,4].
switch2(on) @ [1,3].
switch2(off) @ [3,5].
switch1(on) @ [4, 6].
switch2(on) @ [5,10].
switch1(off) @ [6,8].
switch1(on) @ [8,10].
```

(Integers have been used only for simplicity ; ignore the fact that there are times when the switch is both off and on.)

Now consider the definition

$$\text{light_on} @ T :- \text{switch1(on)} @ T, \text{switch2(on)} @ T .$$

The query

```
?- light_on @ T
```

returns the following answers for T: [1,2], [5,6], [8,10]. The slightly different query defined by

$$\text{light_on} @ T :- \text{switch1(X)} @ T, \text{switch2(X)} @ T .$$

gives the answers: [1,2], [3,4], [5,6], [8,10].

3.3 Negations

The simplest negation to consider is the Prolog negation by failure of the form:

```
not(P @ T).
```

For T indefinite, this can be true only if P is never true. Similarly, for T a definite interval this will succeed only if P is never true during the interval. (The interval T is not changed by such a call.) Thus this construct corresponds to a notion of “never”. This notion has some subtle properties, however, that makes this identification not quite straightforward. As an example, consider the following definition:

```
possible_carrier_of_disease @ T :-  
    exposed_to disease @ T1,  
    T1 =< T,  
    not(tested_negative @ T2),  
    T1 < T2.
```

In the case where exposure occurred during interval $T1$ and no testing is done, `possible_carrier` is true henceforth. But if `tested_negative` is ever true later than $T1$ then `possible_carrier` is not true at any time after $T1$, even before the testing! In cases of open (unfinished) histories this retroactive effect can cause complications in the implementation. A second form of negation, written as

```
not(P) @ T
```

returns successive intervals over which P fails. Since T narrows properly (even though variables in P do not get instantiated), this can be used for “simple” negative conditions. The combination of the two negations can be used to define a notion of “always”:

```
always(P,T) :- not(not(P) @ T).
```

3.4 Events and Causality

Events can be thought of as conditions which last for only short times. One criterion for “short” might be the requirement that the intervals for any two events must be disjoint:

$E1 @ T1, E2 @ T2, \text{ not}(E1@=E2) \rightarrow \text{ not}(T1==T2).$

This would lead to the idea of defining an event class as a maximal set of conditions subject to this law, but this idea will not be pursued further here. Instead, we wish to characterize events in terms of changes of condition. Thus, referring to a previous example, we might wish to define an event of `turning_on_switch1` as a transition from `switch1(off)` to `switch1(on)`.

One way of doing this might be as follows:

```
turning_on_switch1 @ T    :-  
    past(T,P),  
    future(T,F),  
    switch1(off) @ P,  
    switch1(on) @ F.
```

The calls to `past` and `future` serve to create intervals `P` and `F` before and after `T` respectively.

But this definition treats `turning_on_switch` as a derived notion, with the switch condition as a given. A more common case is the one in which events are given and the problem is to determine the appropriate transitions. The definition for this case is a permutation of the above:

```
switch1(on) @ F    :-  
    past(T,P),  
    future(T,F),  
    switch1(off) @ P,  
    turning_on_switch1 @ T.
```

3.5 Partial and Completed Histories

Now imagine the boundary conditions on a history of events given as facts. By boundary conditions we mean an initial set of conditions (whose time intervals can be thought of as extending to negative infinity) and the times of a complete series of external events (that is, events which only appear on the right hand side of rules). By a complete series we mean that the history is now over in the sense that no new events will ever occur; this is, in effect a “closed world assumption”. In the language described above we can formulate many queries over this database of facts, and given a set of

inference rules such as those described above we can make many additional inferences; in particular we can determine the truth value of any predicate over any interval. Predicates or rules involving “never” , of course, depend on the closed world assumption for their justification.

Without altering the logic we can store every conclusion that is drawn by the system so as to eliminate the need for repeated derivation of the same conclusion. This generation of lemmas can be done in any way we choose, e.g. completely haphazardly. However, a natural and systematic way of generating lemmas, given temporal rules, is to proceed systematically from past to future. Ineffect, this simulates the evolution of the system in the order in which the lemmas are generated. A history will be called completed if every lemma/conclusion which can be drawn, has been added to the database; i.e. a maximal set of historical facts given the initial facts and the inference rules.

4 Conclusion

Constraint logic programming with interval arithmetic provides a logical, declarative and executable language in which to express temporal constraint relations. These relations provide the ability to ‘reason’ quantitatively with time intervals on the real line, although not *about* temporal relations as in [1]. Furthermore, the semantic model for interval arithmetic is consistent with semantic models for logic programming as has been shown in [11], which reinforces the suspicion that these kinds of operational temporal operations can be given a clean logical semantics.

References

- [1] Allen, J. F. (1981) “An interval based representation of temporal knowledge”. Proc. 7th Int. Joint Conf. on Artificial Intelligence, Vancouver, B.C., Canada, pp. 221-226.
- [2] Allen, J. F. and Hayes, P. J. “Moments and points in an interval-based temporal logic”, *Computational Intelligence*, **5**, pp.225–238,1989.
- [3] BNR Prolog *User Guide and Reference Manual*, BNR 1988.

- [4] Brown, R. G., Chinneck, J.W. and Karam, G. “Optimization with Constraint Programming Systems” in *Impact of Recent Computer Advances on Operations Research*, North Holland, January 1989.
- [5] Colmerauer, A. “An introduction to Prolog III”, *Communications of the ACM* **33**(7):69, 1990.
- [6] Cleary, J. C. “Logical Arithmetic”, *Future Computing Systems*,**2** (2), pp.125–149, 1987.
- [7] van Hentenryck, P. *Constraint Satisfaction in Logic Programming*, MIT Press, 1989.
- [8] Moore, R. E. (Ed.) *Interval Analysis*, Prentice Hall, New Jersey, 1966.
- [9] Nickel K. L. E. (Ed.) *Interval Mathematics*, Academic Press, New York, 1980.
- [10] Older W. and Vellino A. “Extending Prolog with Constraint Arithmetic on Real Intervals” Proceedings of the Canadian Conference on Electrical and Computer Engineering, 1990.
- [11] Older W, and Vellino A. “Constraint Arithmetic on Real Intervals”, to appear in *Constraint Logic Programming: Selected Research*, A. Colmerauer and F. Benhamou (eds) MIT Press, 1993.
- [12] Sidebottom, G. and Havens, W. “Hierarchical Arc Consistency Applied to Numeric Processing in Constraint Logic Programming” *Technical Report CSSIS TR91-06*, Simon Fraser University, Burnaby, B.C., Canada, 1991.