Transactions on Information and Communications Technologies vol 1, © 1993 WIT Press, www.witpress.com, ISSN 1743-3517

Interactive activity scheduling with object-oriented constraint logic programming

D. Cukierman, R. Ovans & S. Sloseris
Centre for Systems Science, Simon Fraser University, Burnaby, BC, V5A 1S6 Canada

ABSTRACT

An interactive, graphical activity scheduler and its implementation in an object-oriented constraint logic programming language is described. It is known that arc consistency techniques are a tractable decision procedure for the temporal constraint networks examined here. The application described permits the specification of any of Allen's 13 binary temporal constraints on activities and thus subsumes precedence scheduling. Arc consistency and intelligent backtracking are part of the scheduler simply as a by-product of the chosen programming language. The result is a natural, elegant implementation – which supports incremental and reactive scheduling – that required an astonishingly small amount of coding effort. Though the focus is methodology, a comparison of this approach with previous results in temporal constraint satisfaction and representations of temporal reasoning is also included.

INTRODUCTION

A schedule can be succinctly described as a plan that indicates the time and sequence of a set of activities. A great variety of methods have been used to implement computerized scheduling packages, including operations research optimization, constraint satisfaction, and simulation. The role of the completely automated scheduler versus a more interactive alternative has been extensively discussed [10]. Traditional scheduling tools (like PERT and CPM [4]) are restricted to techniques tractable with pencil and paper, while computerized systems can utilize methods that involve extensive search and computation beyond the capability of the human scheduler. The latter approach allows quick analysis of hypothetical situations as well as reactive scheduling (i.e., rescheduling in the event of change).

Transactions on Information and Communications Technologies vol 1, © 1993 WIT Press, www.witpress.com, ISSN 1743-3517

In this paper we describe an interactive, graphical activity scheduler and its elegant implementation in an object-oriented constraint logic programming language. This implementation reveals the utility of a programming language incorporating constraint propagation and intelligent backtracking for the creation of incremental scheduling systems. It is our belief that this system is a step towards the development of scheduling tools that not only resemble those traditionally used by schedulers but also "intelligently" help the user with the task of creating or modifying a schedule. It is debatable that a completely automated expert system is realistic, therefore our focus is an architecture that supports intelligent, interactive, incremental, and reactive scheduling.

BACKGROUND

In this section we discuss the general problem of satisfying temporal constraints and show how activity scheduling is a subset of this general problem. We then introduce the constraint logic programming language Echidna [5], which was used to construct the scheduling tool.

Temporal Constraint Satisfaction

In a recent paper, Meiri has defined the general temporal constraint satisfaction problem (TCSP) as follows [13]. A set of variables represent temporal objects, which are either time points or time intervals. Constraints are of two forms: quantitative or qualitative. All quantitative constraints place absolute bounds on a point (unary constraints), or restrict the temporal distance between two points (binary constraints). All qualitative constraints are disjunctions of binary relations that specify the relative position between any temporal objects (points or intervals). A solution is a consistent assignment of values to the variables such that all the constraints are satisfied.

Meiri's definition is a synthesis of earlier work on constraint-based formalisms for temporal reasoning, namely Allen's interval algebra [1], Vilain and Kautz's point algebra [20], and metric networks [2].

Scheduling problems can be straightforwardly expressed in terms of temporal constraints. As shown in [14], analysis tools like PERT and CPM are a particular case of the TCSP. The precedence relationships typically used in these charts are expressible in terms of the *before* and *meets* relation, two of the 13 relations defined in Allen's interval algebra.

The application described in this paper is restricted to a subset of the general TCSP in that disjunctions of basic relations are not included. However, for scheduling activities in time, only one qualitative constraint (most often the *before* relation) between activities is necessary. This restriction entails that all interval relations are expressible with point-to-point relations [20]. The tempo-

Transactions on Information and Communications Technologies vol 1, © 1993 WIT Press, www.witpress.com, ISSN 1743-3517

ral objects to be assigned values are *activities*. An activity is conceptually an interval of time defined by two time points, a *beginning* and *end*. Allen's interval-to-interval relationships are thus implemented with point-to-point relations on the beginning and end points that define the activities. These interval relations appear in [1] and the associated constraints coincide with that of [14], including `before`'s subsumption of `meets`, a useful simplification for precedence scheduling (see Table 1 where the inverse relationships have been omitted for brevity).

Relation	Constraints
A *before* B	A.end ≤ B.beg
A *meets* B	A.end = B.beg
A *equals* B	A.beg = B.beg, A.end = B.end
A *during* B	A.beg ≥ B.beg, A.end ≤ B.end
A *overlaps* B	A.beg < B.beg, A.end < B.end, A.end ≥ B.beg
A *starts* B	A.beg = B.beg, A.end ≤ B.end
A *finishes* B	A.beg ≥ B.beg, A.end = B.end

Table 1. Temporal Relations Between Activities

Echidna

Constraint logic programming (CLP) [7] is a recent extension to logic programming. CLP embeds within logic programming efficient algorithms for solving constraint satisfaction problems over various domains, while preserving the (desirable) semantic properties of Prolog.

Echidna is a language that combines the paradigms of CLP, object-oriented programming and intelligent backtracking; it constitutes a new approach to the design of expert system shells. Echidna's formation has been influenced significantly by current work in CLP, in particular Van Hentenryck's CHIP system [18]. Like CHIP, Echidna couples constraint propagation with an underlying logic programming language. Both languages inherit the first-order Horn clause semantics of Prolog and employ a k-ary arc consistency algorithm [12] for the propagation of k-ary constraints over discrete domains. It is known that arc consistency exhibits linear time complexity in the number of constraints [13] yet is often much more efficient than search alone. Echidna adds a depth-first incremental derivation scheme and justification-based RMS based on [3] for implementing true intelligent backtracking. In comparison, CHIP inherits both the depth-first search rule and chronological backtracking on failure endemic to Prolog.

Echidna also defines a straightforward formulation of object-oriented programming. The realization of objects within logic programming has seen many proposals (for example [8][19]). In Echidna, objects are instances of syntactic class definitions (called *schemas*) where methods are realized as Horn clauses. We define an object schema as an associated set of predicates and persistent (shared) logical variables. The methods of the object are the predicates and the instance variables of the object are the persistent logical variables. Special syntax is also provided for object creation (`isa`, e.g., `A isa activity`) and message sending (the infix colon operator, e.g., `A:name(excavate)`). Sending a message to an object is equivalent to unifying a goal with some predicate within the object. As well, a message sent to an object – like a query in Prolog – either succeeds or fails. An instance of an object is simply a particular occurrence of the set of persistent logical variables specified in the schema. Access to these variables is via message sending, following normal object-oriented principles.

Allowing for Logical State Change Unlike regular logic programming variables (which are either unbound or instantiated), all the variables in an Echidna program are in one of three states: unbound, bound to a domain, or instantiated (i.e., the domain associated is a singleton). During the execution of a program, a variable could have the following successive values associated:

```
<unbound>
integer
{1..30}
{10..15}
13
```

Such a variable was initially unbound and was successively *refined* (the domain became smaller) until it was instantiated. This succession of state changes is thus a series of monotonic changes. In the current version of Echidna object instance variables may be repeatedly refined towards a ground value but never reassigned once established. The process is monotonic, driven by unification and constraint propagation. Its monotonicity is reversible only through backtracking. Arbitrary state change as a side-effect of method invocation is presently impossible.

The nonmonotonic state change problem that appears in this object-oriented CLP environment is a special case of the general problem of managing state change in an object-oriented logic programming environment (e.g., [8][19]). For this particular application, as it was in [15], the state change mechanism employed is theoretically analogous to *asserting* and *retracting* facts in Prolog. However, given the characteristics of object-oriented CLP, this solution takes a different form. To enact nonmonotonic change in this environment (i.e., when the user wants to change his/her mind about a decision), previ-

Transactions on Information and Communications Technologies vol 1, © 1993 WIT Press, www.witpress.com, ISSN 1743-3517

ously issued goals are *undone*. This solution is viable since new values are not a function of the previous values. Undoing a goal forces the constraint network to assume a state consistent with the goal never having been issued. Very importantly, only changes strictly related to the goal being undone are altered, thanks to Echidna's intelligent backtracking. That is, chronological backtracking does not occur: the undoing of a previous goal does not affect constraints issued chronologically after that goal. This important feature of Echidna permits efficient reactive scheduling.

THE IMPLEMENTATION

The present system permits the creation and modification of activities, the application of quantitative unary constraints on any activity's start time, end time, or duration, and the imposition or elimination of qualitative temporal constraints amongst activities. A window permanently displays a graphical representation of a Gantt chart that reflects the effect of any user action. In the case of conflicting constraints, inconsistencies are presented.

The present implementation runs on UNIX workstations, with an OPEN LOOK graphical user interface. The core of the system is an Echidna knowledge base (program), where the inherent constraint satisfaction problem is incrementally generated and solved. An XView interface coded in C++ communicates with the Echidna knowledge base. The remainder of this section describes in more detail the main components of the system. Example snapshots of the interface are included.

The Knowledge Base

The core of the scheduling system resides in an Echidna knowledge base. The implementation of the knowledge base was a very easy task, requiring less than one person week of effort (cf. [14]) and resulting in a very compact and clean piece of code. This programming paradigm seems highly appropriate for solving this kind of problem. The constraint satisfaction algorithms, the encapsulation of objects, and the intelligent backtracking are all an integral part of the language, thus the programmer is not burdened with these programming tasks.

The basic temporal object and its relations are defined in the class, `timeInterval`. This class has three instance variables: `Begin`, `End` and `Duration`. The `Begin` and `End` variables denote instants of time and are implemented as belonging to a finite contiguous integer domain. This is a design decision consistent with many scheduling tasks, and as pointed out in [4], tables can be used to translate from an integer contiguous subset to any discrete time, including non-contiguous dates. From the standpoint of the implementation language, finite domains permit arc consistency algorithms to enforce local consistency by filtering incompatible values from the domains.

Whenever an object of the `timeInterval` class is created, the initialization method automatically binds the instance variables to their default integer domains and imposes the following constraint:

```
Duration =:= End - Begin.
```

Any subsequent domain refinement to any of these variables will propagate to the other related variables accordingly, thanks to Echidna's constraint satisfaction ingredient. As the user interacts with the system, constraints will gradually refine the domains of these variables to ground values.

Methods within the `timeInterval` class define each of Allen's thirteen temporal relations. For example, the `before` relation is a method that takes as argument an additional object of the `timeInterval` class:

```
before(timeInterval Y):-
     Y:beg(YBeg),
     End =< YBeg.
```

Therefore, issuing a precedence relation between activities is easily implemented as constraining the pertinent instance variables of the activities. Refer again to Table 1 for the complete set of temporal relations and the corresponding implementation.

The class `activity` is a subclass of `timeInterval`, and therefore inherits all of the instance variables and methods defined therein. In the `activity` class, other instance variables are defined (and methods). In this initial version of the scheduler, `Name` is the only additional variable, but other information could be included. Information pertaining to resource consumption by an activity could be modelled with additional instance variables and possibly other class definitions. Resource allocation has not been implemented yet, but job-shop scheduling and crew assignment constitutes an extension of the present system under study at the moment.

Activities are created by issuing goals of the following form:

```
A isa activity.
B isa activity.
```

To impose that activity *A* complete before activity *B* starts, the goal to issue is:

```
A:before(B).
```

User Interface

A useful feature of Echidna is its definition and implementation of a protocol that permits communication with an external C++ process through TC/IP sock-

Transactions on Information and Communications Technologies vol 1, © 1993 WIT Press, www.witpress.com, ISSN 1743-3517

ets. Goals can be issued or undone via this protocol. In this instance, the external process defines an interface for the construction of project schedules.

External methods are defined in the Echidna knowledge base as special predicates that communicate with the interface. The following extract illustrates an external method `(dataAc)` invoked from the initialization method of the `activity` class:

```
activity:-
     Duration =:= End - Begin,
     dataAc(Begin,End,Duration).
```

The external method serves as a communication channel; any update to a variable appearing as an argument in this external method is automatically captured in the external C++ program and rendered in the interface. This occurs whenever the variables appearing as arguments in the external method are changed, either through the issuing of a goal, backtracking, or constraint propagation.

Examples

The system is interactively guided by the user, who can create, modify or delete activities and temporal relations between activities. As soon as the user provokes any change, all the constraints are propagated and the corresponding results are displayed in a Gantt chart.

Figure 1 shows the beginning stages of a schedule for the construction of a house. Directly beneath the name of each activity is the current domain for the beginning, end, and duration time points respectively. Note that a fixed duration, though likely the norm, is not necessary. Some activities (like *Excavate*) have a fixed beginning as well as a fixed duration. Other activities only have the duration fixed, and therefore have a tail following the rectangle. This line indicates its *float* or *slack time*, i.e. the difference between the latest ending and the earliest ending.

Figure 2 illustrates the dialogue boxes used to create and modify activities and relations. Note that within an activity any point can be quantitatively constrained to belong to a particular interval. Maintenance of relations is achieved in a user-friendly way. As relations are added or deleted, the activities "move" in the Gantt chart accordingly.

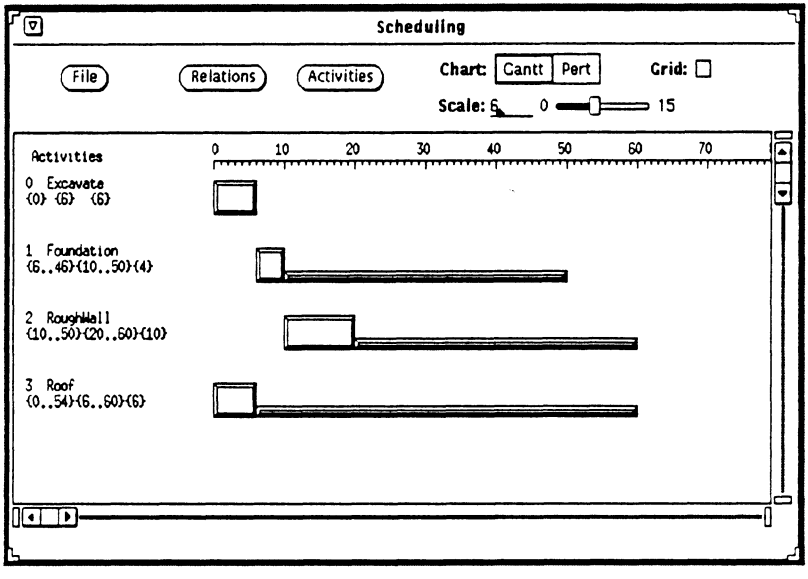


Figure 1. Gantt chart showing the current state of the activities.

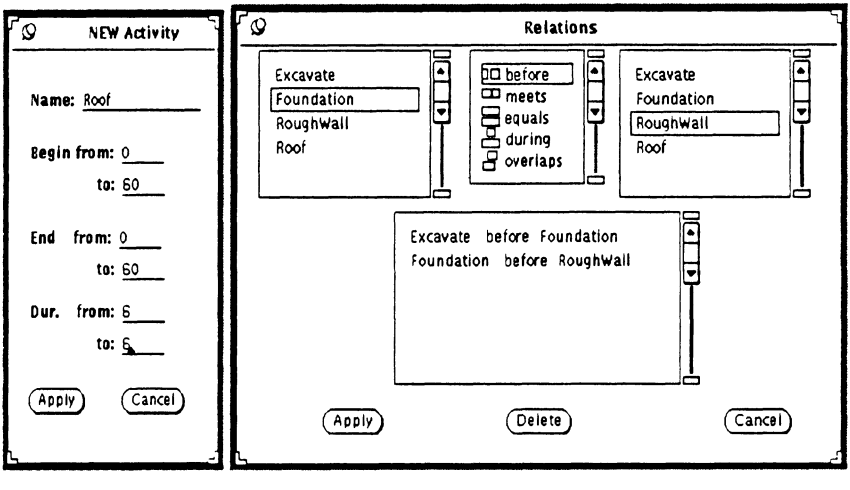


Figure 2. Activity *Roof* is created. Two relations have already been added to the schedule.

Figure 3 is a PERT chart for a house building project taken from [6] and used in [14]. This very same example has been included in the present paper as a means of showing how the system works and for comparison with the system described in [14].

Figure 4 shows the corresponding Gantt chart when all the constraints have been added. The relations are all *before* constraints respecting the PERT chart of Figure 3. Also, two activities have their finishing times fixed which causes all activities to be scheduled. A difference is noted with the result obtained in [14] and is accounted for by an absent constraint in [14] with respect to the original problem.

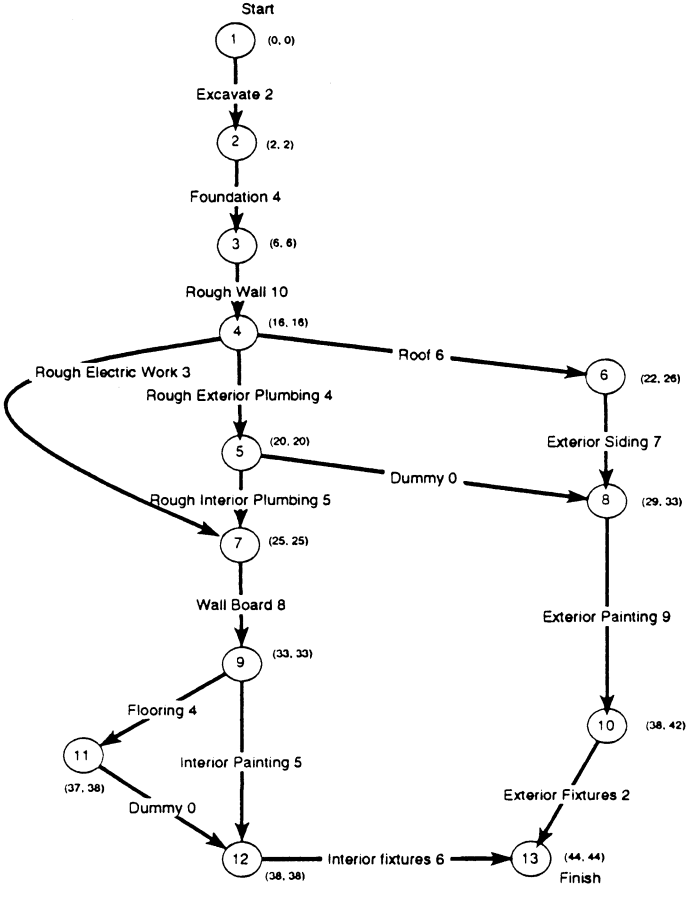


Figure 3. PERT chart for the sample project.

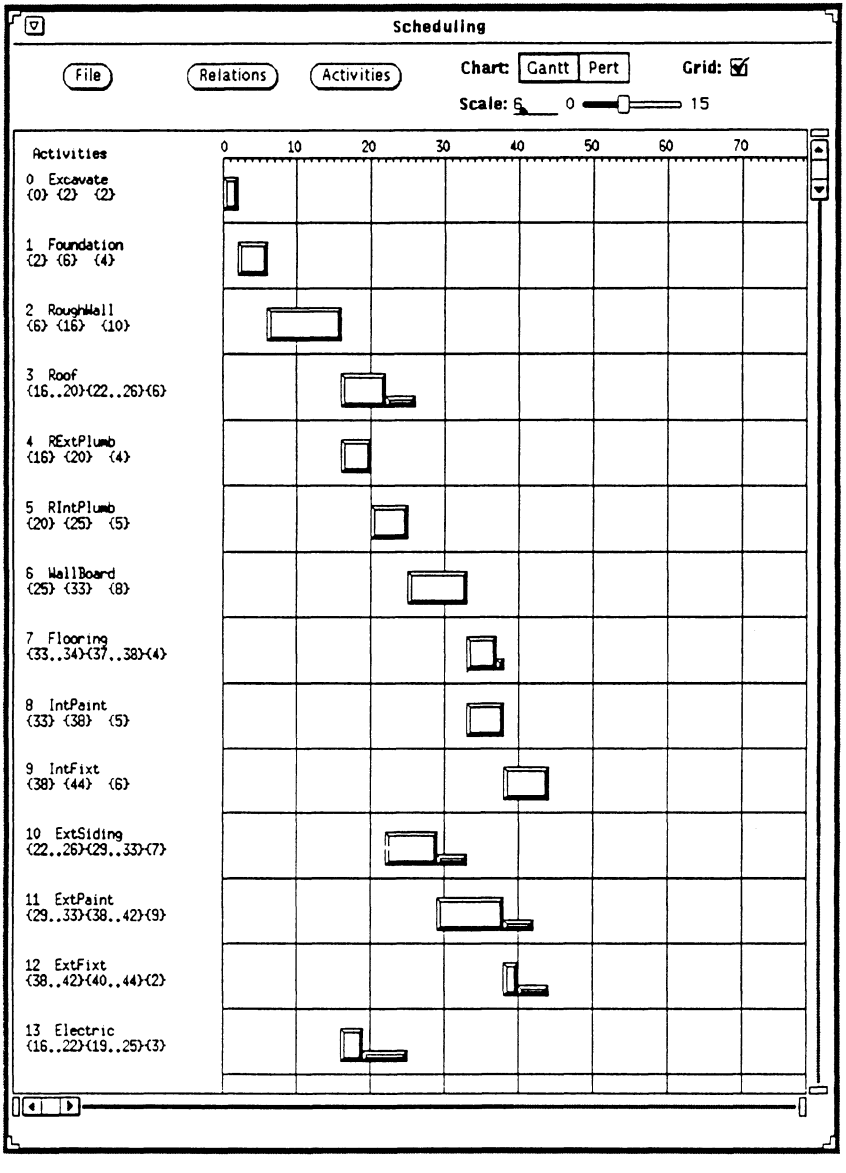


Figure 4. *IntFixt* and *ExtFixt* have their end times fixed to the earliest date possible. This schedule corresponds to the example shown in [14]. Note that [14] shows *Flooring* with a slack time which is longer than that shown here; the dummy precedence from nodes 11 to 12 (in the PERT chart, Figure 3) is not respected in their case.

Conflicts

Figure 5 further continues this example with the addition of an inconsistent constraint. Given the intelligent backtracking feature of Echidna, the conflict set reported does not include goals irrelevant to the conflict, irrespective of the chronological order in which they were issued.

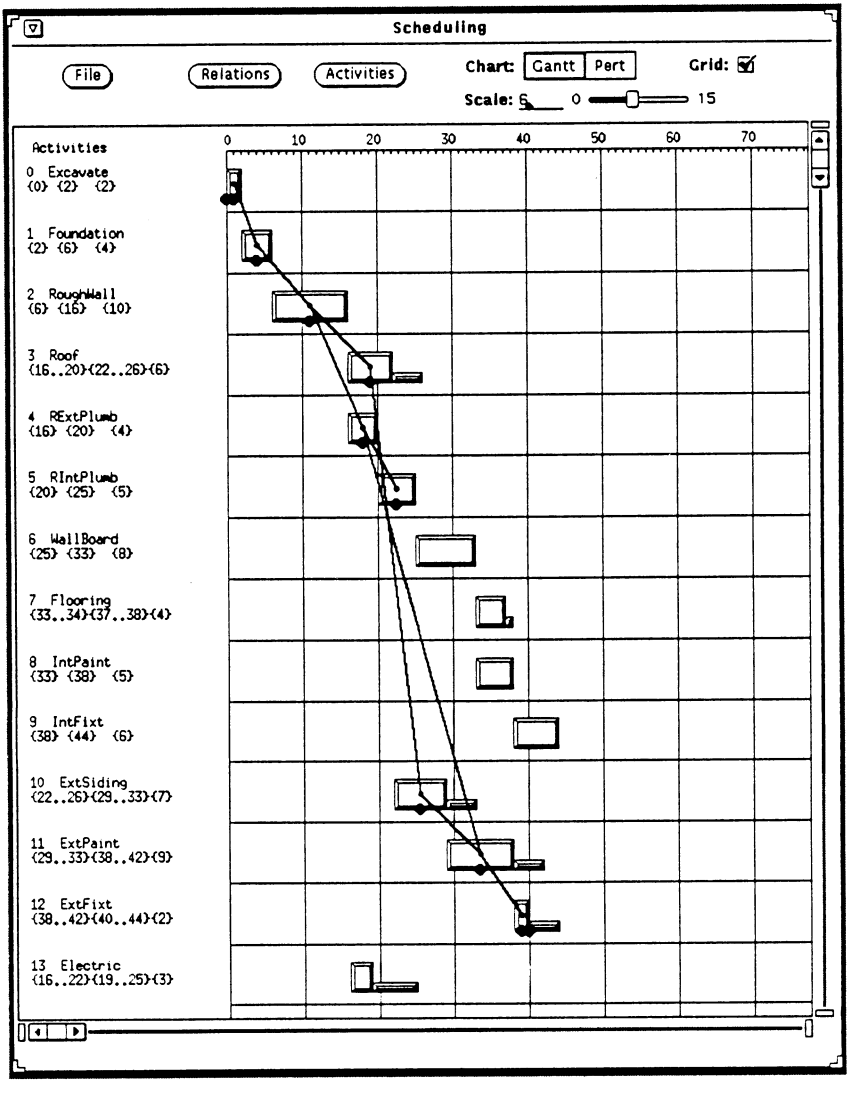


Figure 5. *ExtPaint before Excavate* creates a conflict. The activities and the relations involved are signalled graphically.

The chart displayed is in fact a combination of Gantt and PDM (precedence diagram method) charts. This new kind of chart shows – like the Gantt chart – the relative lengths of the activities and their associated float (if any). Inspired by PDM charts, it also depicts graphically the relations in conflict, which are the edges in this graph. Moreover, dots indicate points or durations that have been fixed to a single value and participate in the conflict. An option allows the user to combine the Gantt and PDM charts to show the activities and all the relations between them regardless of conflict.

When conflicts arise, there are no predefined rules to relax constraints, rather, the user will take the appropriate decision given the information provided by the system and his/her own expertise. It is the user who decides whether he/she wants to relax a previous constraint, the constraint just added, or modify a particular value for some activity.

Interactive and Automatic

It should be noted that individual, interactive modifications can instantiate an entire project. This was the case in Figure 4: the schedule was "solved" by restricting the end of both *IntFixt* and *ExtFixt*. As well, a *Finish* dummy activity could have been created, as the example in [14] suggests, and the end of this single activity could be restricted. Such a *Finish* activity could act as a vehicle for hypothetical exploration by trying different values for its end time.

Moreover, the system can automatically find solutions by following certain directives. A goal can be issued to the Echidna reasoner so that it consistently instantiates the beginning of every activity to the earliest (or latest) possible time. The predicates `mindomain` and `maxdomain` of Echidna accomplish the instantiation of any time point to the minimal or maximal possible value respectively, among the possible values for that point, given the constraints already imposed.

DISCUSSION

In this section we first compare our implementation with the theory that has appeared in the literature and then consider possible extensions to the system.

Temporal Ontology: Charts and Formalisms

Formalisms to represent time deal with *time points* [20][2], *time intervals* [1][17] or a combination of the two [13][9] as basic temporal objects. Similarly, scheduling charts reflect intervals (Gantt, PDM), timepoints (PERT, Milestone) or a combination of the two. Orthogonal to the point versus interval debate is the philosophy espoused in the various charts that have been used in scheduling: some charts stress the relations among the temporal objects and some do

Transactions on Information and Communications Technologies vol 1, © 1993 WIT Press, www.witpress.com, ISSN 1743-3517

not. For example, the Gantt chart reflects the intervals during which each activity occurs, but with no relations among them. In a similar way Milestone charts reflect important time points in a project, most often corresponding to the finishing times of activities, but without relationships expressed between them.

Charts that stress the relations between the different activities include the traditional PERT chart and also the PDM. In the scheduling literature, PERT charts are networks where the nodes are time points and the edges between them are the activities. This is comparable to the metric constraint networks of [2]. PDM charts, which have the activities represented as nodes in the network and the edges between them indicating the qualitative relationship amongst them, are more akin to Allen's algebra and the implementation described here.

It would seem that the issues in reasoning about time converge to questions surrounding the representation of the basic temporal objects and the constraints said representations permit. As mentioned earlier, the basic temporal object of our system is an interval defined by its extreme end points. However, given that the beginning and end points range over a domain and that the duration of an interval can be set to zero, these intervals can just as well represent time points. Implementing points as zero length intervals is one option, but others are possible and under study, as for example defining a point class which would be more akin to time ontological issues [1][17].

In terms of constraints, therefore, the developed system permits qualitative constraints between intervals, points, and intervals and points, and quantitative constraints on and between points. Presently, qualitative constraints are restricted to any conjunction of the 13 basic binary temporal relations defined in Allen's interval algebra. Qualitative constraints on intervals are implemented by constraining their ending points, as shown in Table 1 . Unary quantitative constraints are possible over the discrete domains of the beginning and end points, and binary quantitative constraints are implemented naturally with the `Duration =:= Begin - End` constraint. To be able to constrain the distance between any two points (for example, the beginning of one activity with respect to the ending of another) a dummy activity that *meets* these two points can be defined.

Given that time intervals are implemented as extreme points constrained to a particular domain, and all constraints are implemented as convex point-to-point relations, the constraint network generated by the Echidna reasoner for this application is a special case of Meiri's augmented qualitative network [13] restricted to conjunctions of qualitative constraints. Significantly, this subset represents a tractable case of the general TCSP: arc consistency is a necessary and sufficient condition of satisfiability for convex point-to-point qualitative networks [13].

Transactions on Information and Communications Technologies vol 1, © 1993 WIT Press, www.witpress.com, ISSN 1743-3517

Future Enhancements

One line of future development would see the establishment of constraint priorities. This priority information could be used in the event of a conflict to present the user with guidance in deciding which constraint to relax. Note that depending on the action the user takes, conflicts may still remain after a constraint is undone. In such a case, the system then reveals the remaining constraints. The system therefore is sound, but could be considered incomplete in that when signalling a conflict, other conflicts may exist but go unreported. Further research into the question of conflict reporting is thus required.

Another potential enhancement is to add more generality to the allowable temporal constraints so that for example disjunction of relations can be expressed. Given the logic programming foundation of Echidna, disjunctive constraints are easily achieved via nondeterministic methods. For example, *before or after* can be expressed as:

```
disjoint(B):- before(B).
disjoint(B):- after(B).
```

However, it is questionable whether or not it is desirable to add more expressive power to the system since the penalty is a loss of tractability when disjunctions are introduced [20][13].

In the same vein, quantitative constraints could be improved. At present, time points can be constrained to a single domain, and again, disjunction of constraints where a time point is constrained with inequalities is conceivable. The power to impose these constraints already exists in the Echidna reasoner; it is only a matter of redesigning the interface to support the concept.

An alternative offered by the Echidna language is implementing an interval as a single variable constrained to lie within an interval of real numbers. Since Echidna also implements a form of arc consistency over real intervals [16], the computation power would not be compromised.

An improvement to the graphical interface would be inclusion of the ability to modify any of the variables associated with an activity by simply dragging and dropping or resizing the associated rectangle. Thus the Gantt chart would operate as both an input and output device, making the editor a more interactive tool.

In a completely different direction, the system could become a general constraint editor, serving as a visual semantic spreadsheet that permits the creation of objects and the imposition of constraints, obtaining information from the class definitions appearing in an arbitrary Echidna knowledge base.

Transactions on Information and Communications Technologies vol 1, © 1993 WIT Press, www.witpress.com, ISSN 1743-3517

ACKNOWLEDGEMENTS

The authors wish to thank Bill Havens and Miron Cuperman of the Expert Systems Lab at Simon Fraser University for fruitful discussions and Echidna programming support respectively. Diana Cukierman gratefully acknowledges the financial support of the Canadian International Development Agency. Russell Ovans gratefully acknowledges the support of PRECARN Associates Inc.

REFERENCES

1. Allen, J.F. 'Maintaining Knowledge About Temporal Intervals', *Communications of the ACM*, Vol. 26 (11), pp. 832-843, 1983.

2. Dechter, R., Meiri, I. and Pearl, J. 'Temporal Constraint Networks', *Artificial Intelligence*, Vol. 49, pp. 61-95, 1991.

3. Doyle, J. 'A Truth Maintenance System' *Artificial Intelligence*, Vol. 12, pp. 231-272, 1979.

4. Fleming, Q. W., Bronn, J. W. and Humphreys, G. C. *Project and Production Scheduling*. Probus Publishing Co., Chicago, 1987.

5. Havens, W.S., Sidebottom, S., Sidebottom, G., Jones, J. and Ovans, R. 'Echidna: a Constraint Logic Programming Shell', *Proceedings of the Second Pacific Rim International Conference on Artificial Intelligence*, Seoul, Korea, pp. 165-171, 1992.

6. Hillier, F. S. and Lieberman, G. J. *Introduction to Operations Research*. Fourth edition. Holden-Day, Oakland, 1986.

7. Jaffar, J. and Lassez, J.L. 'Constraint Logic Programming', *Fourteenth ACM Symposium on the Principle of Programming Languages*, pp. 111-119, 1987.

8. Kahn, K. 'Objects – A Fresh Look', *Proceedings of the European Conference on Object-Oriented Programming*, pp. 207-223, 1989.

9. Kautz, H. A. and Ladkin, P. B. 'Integrating Metric and Qualitative Temporal Reasoning', *Proceedings Ninth National Conference on Artificial Intelligence*, pp. 241-246, 1991.

10. Kempf, K., Le Pape, C., Smith, S. F. and Fox, B. R. 'Issues in the Design of AI-Based Schedulers: A Workshop Report', *AI Magazine*, Vol. 11(5), pp. 37-55, 1991.

11. Mackworth, A. 'Consistency in Networks of Relations', *Artificial Intelligence*, Vol 8, pp. 118-126, 1977.

Transactions on Information and Communications Technologies vol 1, © 1993 WIT Press, www.witpress.com, ISSN 1743-3517

12. Mackworth, A. and Freuder, E. 'The Complexity of Some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems', *Artificial Intelligence*, Vol. 25, pp. 65-74, 1985.

13. Meiri, I. 'Combining Qualitative and Quantitative Constraints in Temporal Reasoning', *Proceedings Ninth National Conference on Artificial Intelligence*, pp. 260-267, 1991.

14. Meng, A. C. and Raja, B.A. 'Logos TCS: An Expert System Language for Operations Management based on Temporal Constraint Satisfaction', *Proceedings of the Sixth Conference on Artificial Intelligence Applications*, pp. 215-221, 1990.

15. Ovans, R. and Davison, R. 'An Interactive Constraint-Based Expert Assistant for Music Composition', *Proceedings of the Ninth Canadian Conference on Artificial Intelligence*, Vancouver, Canada, pp. 76-81, 1992.

16. Sidebottom, G. and Havens, W. 'Hierarchical Arc Consistency for Disjoint Real Intervals in Constraint Logic Programming', *Computational Intelligence*, Vol. 8 (4), pp. 601-623, 1992.

17. Shoham, Y. *Reasoning About Change: Time and Causation from the Standpoint of Artificial Intelligence*. MIT Press, Cambridge and London, 1988.

18. Van Hentenryck, P. *Constraint Satisfaction in Logic Programming*. MIT Press, Cambridge and London, 1989.

19. Vaucher, J., Lapalme, G. and Malenfant, J. 'SCOOP: Structured Concurrent Object Oriented Prolog', *Proceedings of the European Conference on Object-Oriented Programming*, pp. 191-211, 1988.

20. Vilain, M. and Kautz, H. 'Constraint Propagation Algorithms for Temporal Reasoning', *Proceedings of the National Conference on Artificial Intelligence*, Philadelphia, USA, pp. 377-382, 1986.