Association for Information Systems
AIS Electronic Library (AISeL)

AMCIS 1998 Proceedings

Americas Conference on Information Systems (AMCIS)

December 1998

Constraint Propagation in Workflow Systems Using CLIPS

Susan Chinn
Penn State Erie

Gregory Madey
Kent State University

Constraint Propagation in Workflow Systems Using CLIPS

Susan J. Chinn
The Behrend College School of Business
Penn State Erie

Gregory R. Madey
College of Business Administration
Kent State University

Constraint Propagation

Workflow applications are involved with sequencing activities, monitoring and routing jobs, and coordinating work schedules. These applications must often track changing information about temporal relationships, and maintain a correct set of dependencies among those relationships during the workflow (Egger and Wagner, 1992). Constraint propagation provides a mechanism for updating temporal relationships. Research in temporal reasoning has focused on the problem of constraint propagation, especially for interval-based temporal relations (Allen, 1983; Kautz and Ladkin, 1991). Allen's model of interval-based temporal relations (1983) uses a transitivity table to infer sets of possible relations among any three events. For example, if we know that A occurs *before* B, and B occurs *during* C, then A and C may participate in any of the following relations: *before*, *overlaps*, *meets*, *during*, and *starts*. We do not know which relation is correct, however, until we receive additional information about A and C; we have only a set of constraints on the possible truthfulness of the system's state. As new information is introduced in the form of additional temporal relations (e.g., a work schedule being "firmed up"), the network of temporal constraints needs to be re-computed and checked for correctness. Thus new information about A and C means that previous relations produced from the transitivity table can be eliminated, and relations between C and B need to be re-computed.

Scheduling applications that use expert systems depend on constraint propagation to maintain consistency in the schedules (Dorn, 1992; Brown, Marin, and Scherer, 1995). Most of these applications use expert system tools that have a proprietary temporal reasoning component. Temporal constraint propagation, however, can be implemented with a non-proprietary forward-chaining expert system tool, where additional incoming facts can be used to constrain previously derived temporal relations. In this paper, we use CLIPS (C Language Integrated Production Language) to show that Allen's algorithm for constraint propagation can be implemented through a series of programs. The constrained temporal network can be used to maintain a set of current information about the relationships among any three interval events.

Implementation Using CLIPS

We use a workflow example based on an engineering design change process with interval relations to represent scheduled work periods among engineers MA, MB, and MC (Chinn & Madey, 1997). Our scenario is based on Allen's example of three interval relations (1983) to ensure that our implementation of the constraint propagation algorithm produces the same results. We design three programs in CLIPS to implement constraint propagation. The first program takes a set of facts constructed as interval relations and applies Allen's transitivity table to them. An interval relation between MB and MC is compared to interval relations between MA and MB to determine new temporal relations between MA and MC. Figure 1 shows one of the rules used to compute the new set of relations. The resulting set of relations is added to the fact-list and written to a file for use by the second program. The second program's purpose is to handle any new temporal relations that supply additional information that introduce new temporal constraints. We add three new relations between MA and MC: *overlaps*, *starts*, and *during*. Before this program is run, fact duplication in CLIPS must be allowed, because two of these relations have already been computed as possible constraints. Once the new facts are added, any duplicates are detected by comparing two facts that are identical in every way except for the fact-index. Those relations that are duplicated form the intersection of old and new relations between MA and MC. One duplicate set is retained, and other relations between MA and MC that are not duplicated are deleted, because they are not in the intersection, and represent "impossible" relationships (Allen, 1983). Figure 2 shows the rules that check for duplicate facts and eliminate other relations. Once this program has ended, fact-duplication is disabled to prevent redundant or unnecessary duplications when the network is re-computed. The third program prepares the facts saved by the second program for the next iteration of constraint propagation with the new constraints. At this point, we have three sets of relations: those between MA and MC, those between MB and MC, and those between MA and MB. The original constraints between MB and MC are temporarily "removed" by reassigning them to a new deftemplate called "old-set." They will be needed later when we compute the intersection between the old and new sets. In order to obtain the new set of MB and MC relations, the relations

```
/* original set of relations */
(relation (type before) (r1 MB) (r2 MC) (dup nil) (new nil))
(relation (type meets) (r1 MB) (r2 MC) (dup nil) (new nil))
(relation (type met-by) (r1 MB) (r2 MC) (dup nil) (new nil))
(relation (type after) (r1 MB) (r2 MC) (dup nil) (new nil))
(relation (type overlapped-by) (r1 MA) (r2 MB)(dup nil) (new nil))
(relation (type met-by) (r1 MA) (r2 MB) (dup nil) (new nil))

;; rule to compute transitivity between “overlapped-by” and “before” relations
(defrule oi-before
      (relation (type overlapped-by) (r1 ?r1) (r2 ?r2))
      (relation (type before) (r1 ?r2) (r2 ?r3))
=>
(assert (relation (type before) (r1 ?r1) (r2 ?r3)))
(assert (relation (type overlaps) (r1 ?r1) (r2 ?r3)))
(assert (relation (type meets) (r1 ?r1) (r2 ?r3)))
(assert (relation (type contains) (r1 ?r1) (r2 ?r3)))
(assert (relation (type finished-by) (r1 ?r1) (r2 ?r3))))

/* relations derived from first program */
(relation (type before) (r1 MA) (r2 MC) (dup nil) (new nil))
(relation (type overlaps) (r1 MA) (r2 MC) (dup nil) (new nil))
(relation (type meets) (r1 MA) (r2 MC) (dup nil) (new nil))
(relation (type contains) (r1 MA) (r2 MC) (dup nil) (new nil))
(relation (type finished-by) (r1 MA) (r2 MC) (dup nil) (new nil))
(relation (type after) (r1 MA) (r2 MC) (dup nil) (new nil))
(relation (type starts) (r1 MA) (r2 MC) (dup nil) (new nil))
(relation (type started-by) (r1 MA) (r2 MC) (dup nil) (new nil))
      (relation (type equals) (r1 MA) (r2 MC) (dup nil) (new nil))
```

Figure 1. One Rule from the Transitivity Table Used to Generate the New Set of Relations in the First Program

between MA and MB must be inverted so that they can be compared with MA and MC relations through transitivity by re-running the first program. Figure 3 shows the rules that save the old set and prepare the new set to be inverted.

When the first program is run again, three new relations between MB and MC are found from the transitivity table. They must then be compared with the old set of relations MB and MC to find the intersection. The second program then converts the original set from the *old-set* deftemplate facts to *relation* deftemplate facts, and computes the intersection as before. The end result is the new intersection of MB and MC, which, with the relations between MB and MA, and MA and MC, form the final network.

Conclusion

We have shown how an expert system shell, CLIPS, which cannot perform temporal reasoning “automatically” (Riley, 1996), can perform constraint propagation using Allen’s algorithm for temporal intervals. The programs could be incorporated as a component in a workflow system to maintain a consistent representation of the work schedules among engineers in a review process.

References

References are available upon request from the authors (sjc6@psu.edu; gmadey@synapse.kent.edu).

```
/* three new relations asserted */
(assert (relation (type overlaps) (r1 MA) (r2 MC)))
(assert (relation (type starts) (r1 MA) (r2 MC)))
(assert (relation (type during) (r1 MA) (r2 MC)))

;; if there are duplicates, note and save one set, and delete the others
(defrule dup-facts
      ?f1 <- (relation (type ?type1) (r1 ?r1) (r2 ?r2))
      ?f2 <- (relation (type ?type2&?type1) (r1 ?r3&?r1) (r2 ?r4&?r2) (dup ?dup))
      (test (neq ?f1 ?f2))
=>
(retract ?f1)
(modify ?f2 (dup dup))
(printout t "duplicate fact deleted" crlf))

;; delete anything not in intersection
(defrule delete-not-intersect
      (declare (salience -10))
      ?f1 <- (relation (type ?type1) (r1 ?r1) (r2 ?r2) (dup dup))
      ?f2 <- (relation (type ?type2&~?type1) (r1 ?r3&?r1) (r2 ?r4&?r2) (dup nil))
=>
(retract ?f2)
(printout t "Deleting facts not in intersection" crlf))

/* relations left at the end of the second program */
(relation (type before) (r1 MB) (r2 MC) (dup nil) (new nil))
(relation (type meets) (r1 MB) (r2 MC) (dup nil) (new nil))
(relation (type met-by) (r1 MB) (r2 MC) (dup nil) (new nil))
(relation (type after) (r1 MB) (r2 MC) (dup nil) (new nil))
(relation (type overlapped-by) (r1 MA) (r2 MB) (dup nil) (new nil))
(relation (type met-by) (r1 MA) (r2 MB) (dup nil) (new nil))
(relation (type starts) (r1 MA) (r2 MC) (dup dup) (new nil))
(relation (type overlaps) (r1 MA) (r2 MC) (dup dup) (new nil))
```

Figure 2. After Adding Three New Relations, Additional Constraints Are Inferred from the Second Program

```
;; temporarily “remove” old relations
(defrule delete-old-relations
      (declare (salience 100))
      ?f1 <- (relation (r1 ?r1) (r2 ?r2) (new nil))
      ?f2 <- (relation (type ?type) (r1 ?r2) (r2 ?r3) (new nil))
=>
(retract ?f2)
(assert (old-set (type ?type) (r1 ?r2) (r2 ?r3))))

;; choose one set to invert so that transitivity table will work(
defrule prepare-new-set
      ?f1 <- (relation (type ?type1) (r1 ?r1) (r2 ?r2))
      ?f2 <- (relation (type ?type2) (r1 ?r1) (r2 ?r3&~?r2))
=>
(retract ?f2)
(assert (toinvert (type ?type2) (r1 ?r1) (r2 ?r3))))

;; take the inverse of a relation, “before” in this case (defrule convert-before
      (declare (salience -10))
      ?i <- (toinvert (type ?type&before) (r1 ?r1) (r2 ?r2))
=>
(retract ?i)
(assert (relation (type after) (r1 ?r2) (r2 ?r1) (new new))))

/* resulting relations */
(relation (type starts) (r1 MA) (r2 MC) (dup dup) (new nil))
(relation (type overlaps) (r1 MA) (r2 MC) (dup dup) (new nil))(old-set (type after) (r1 MB) (r2 MC))
(old-set (type met-by) (r1 MB) (r2 MC))(old-set (type meets) (r1 MB) (r2 MC))(old-set (type before) (r1 MB) (r2 MC))(relation (type
overlaps) (r1 MB) (r2 MA) (dup nil) (new new))
(relation (type meets) (r1 MB) (r2 MA) (dup nil) (new new))
```

Figure 3. Preparing for the Next Iteration