International Journal of Knowledge-based and Intelligent Engineering Systems 17 (2013) 209–217
DOI 10.3233/KES-130251
IOS Press

Specifying and solving symbolic and numeric temporal constraints

Samira Sadaoui*, Malek Mouhoub and Xiao Feng Li
Department of Computer Science, University of Regina, Regina, SK, Canada

Abstract. Representing and solving combinatorial problems, especially those including temporal constraints, using a constraint programming language remains a challenging task. In this paper, we present a tool to assist users in specifying and solving problems under qualitative and quantitative temporal constraints. The tool is based on the TemPro framework that has the ability to manage both numeric and symbolic temporal constraints within a unique model. Our tool provides a generic template that can be specialized to describe a wide variety of temporal constraint applications. Given a problem under temporal constraints, the proposed tool with its friendly graphical user interface first assists the user in the different steps of the problem specification. The graph representation of the temporal constraint problem and its consistent scenarios are then automatically generated and visualized during the solving phase. The user has also the ability to add or remove some constraints and see the effects of these changes on the consistency of the problem.

Keywords: Constraint satisfaction, object constraint language, temporal reasoning

1. Introduction

A Constraint Satisfaction Problems (CSP) consists of a finite set of variables with finite domains, and a finite set of constraints restricting the possible combinations of variable values [6,10]. A solution tuple to a CSP is a set of assigned values to variables that satisfy all the constraints. Since a CSP is known to be an NP-hard problem in general, a backtrack search algorithm of exponential time cost is required to find a complete solution. In order to overcome this difficulty in practice, constraint propagation techniques have been proposed [6,9,12,13]. The goal of theses techniques is to reduce the size of the search space before and during the backtrack search. In a previous work [14,16], a CSP-based model, called TemPro, has been developed for representing and solving temporal CSPs involving numeric and symbolic temporal constraints. More precisely, TemPro translates an application involving temporal information into a binary CSP[1] where variables are temporal events defined on domains of numeric intervals and binary constraints between variables correspond to disjunctions of Allen primitives [1]. We call this latter a Temporal CSP (TCSP).[2]

Many real-life applications, including scheduling, planning and temporal databases, can be considered as instances of a TCSP. Formulating these problems as TCSPs and solving them can however be very challenging tasks. The latter require a strong knowledge in temporal reasoning and CSP search techniques. To address these issues, we propose in this paper a tool to assist users in specifying and solving temporal constraint applications. The tool is based on the model TemPro and integrates its related solving techniques [14,16] through the Choco constraint solving library [4]. In-

*Corresponding author: Samira Sadaoui, Department of Computer Science, University of Regina, Regina, SK, Canada. E-mail: sadaouis@uregina.ca.

[1] A binary CSP is a CSP where all the constraints are either unary or binary relations between variables.

[2] Note that this name and the corresponding acronym was used in [7]. The TCSP, as defined by Dechter et al., is a quantitative temporal network used to represent only numeric temporal information. Nodes represent time points while arcs are labeled by a set of disjoint intervals denoting a disjunction of bounded differences between each pair of time points.

ISSN 1327-2314/13/$27.50 © 2013 – IOS Press and the authors. All rights reserved

deed, since Choco is open source, therefore we extend its class library with the TCSP algorithms mentioned in this paper. Assisting the user starts by providing him with a generic Object Constraint Language (OCL)-based template that can be instantiated to describe temporal problems as TCSPs. We choose OCL as it is a well-defined and flexible constraint description language for object-oriented models, and OCL is easier to learn than the Choco language. Nevertheless, OCL lacks the description for temporal constraints, so we extend it with new keywords to be able to handle TCSP problems. Now users just need to represent their problems and solution rules with the OCL extensions, and the resulting TCSP specifications are easier to read and understand that the corresponding Choco code. The assistance also includes the checking of the consistency of the whole TCSP specification corresponding to a given combinatorial problem, as well as selecting the most appropriate solving algorithm for the current TCSP problem. Selecting the right solving algorithm is another challenging task since most users cannot decide which one best suits their specific applications. Thanks to our tool, users do not have to learn constraint programming languages, such as Choco, and the TCSP solving techniques. Furthermore, our tool is implemented with a generic software architecture capable of modelling a wide variety of problems under temporal constraints. While it currently relies on the Choco library [4], it is designed in a way that it can be coupled with any other existing constraint class library, such as ILOG [11], Prolog [8], Java Cream [5], . . . etc. Consequently, the tool can be enhanced any time with new constraint technologies and solving algorithms.

In addition to assisting non experts for solving temporal constraint applications, our tool with its friendly front end Graphic User Interface (GUI) is very useful in academia. Indeed, starting from the TCSP template, the tool will first take the user through the different steps of the representation phase. Using the text and/or the graphic modes, the user will then have the ability to interact with the tool by asking for one or more consistent scenarios/solutions, changing the description of the problem and see how does this change affects the consistency of the problem. Moreover, the tool will provide this latter with the different steps of the solving process before returning the final answer.

Handling temporal constraints has been addressed by some of the constraint solvers in the past two decades. However these temporal constraints are often limited to numeric relations between time instants such as in ILOG scheduler [11] and Choco [4]. A solver tool including a friendly GUI, and called GSTP, has been recently developed to handle time granularities together with numeric temporal constraints [3]. Based on the STP framework [7], these latter binary relations have the form `Y -X 2 [m, n]G`, where `m` and `n` are respectively the min and max values of the distance between `X` and `Y` in terms of time granularity `G`. Finally, an implementation of a temporal constraint solver with a GUI has been developed in [20] to manage and visualize only symbolic temporal constraints modeled using the Allen Algebra [1]. Comparing to the above systems, the particularity of our solver is its ability to handle both numeric and symbolic temporal constraints which are often present together in the same temporal constraint problem. Moreover, our solver comes with a friendly GUI that allows a better assistance and interaction with the end-users.

The paper is organized as follows. Section 2 introduces, through a temporal problem, the TemPro model and its underlying solving algorithms. Section 3 first presents the OCL extensions that are necessary for describing TCSP applications as well as the generic TCSP template based on the proposed OCL extensions. In addition, Section 3 shows how to specialize the TCSP template to specify two temporal reasoning problems. Section 4 exposes the generic software architecture of our specification and solving tool, and also provides an example of an application. Section 5 concludes with a summary and future work.

2. Managing temporal constraints: The model TemPro

Example 1. Let us consider the following temporal reasoning problem:

> John, Mary and Wendy *separately* rode to the soccer game. It takes John *30 minutes*, Mary *20 minutes* and Wendy *50 minutes* to get to the soccer game. John either *started or arrived* just as Mary *started*. John either *started or arrived* just as Wendy *started*. John left home *between 7:00 and 7:10*. Mary and Wendy *arrived together but started at different times*. Mary arrived at work *between 7:55 and 8:00*. John's trip *overlapped* the soccer game. Mary's trip took place *during* the game or else the game took place *during* her trip. The soccer game *starts at 7:30* and *lasts 105 minutes*.

Using the modelling framework TemPro [14,16], the problem above is translated to the TCSP represented by

Table 1
Allen primitives

Relation	Symbol	Inverse	Meaning
X Before Y	B	Bi	X Y
X Equals Y	E	E	X Y
X Meets Y	M	Mi	X Y
X Overlaps Y	O	Oi	X Y
X During Y	D	Di	X Y
X Starts Y	S	Si	X Y
X Finishes Y	F	Fi	Y X

the graph in Fig. 1. The nodes of the graph correspond to the four events of our story, namely: John, Mike and Wendy are going to the soccer game and the soccer game itself. Events have here a uniform reified representation made up of a proposition and its temporal qualification: $Evt = OCCUR(p, I)$ defined by Allen [1] and denoting the fact that the proposition p occurred over the interval I. For the sake of notation simplicity, an event is used in this paper to denote its temporal qualification. The domains of the four events are the possible time intervals each event can take. Each event domain is expressed by the fourfold [*begintime, endtime, duration, step*] where *begintime* and *endtime* are respectively the earliest start time and the latest end time of the corresponding event, *duration* is the duration of the event and *step* defines the distance (number of time units) between the starting time of two adjacent intervals within the temporal window. This will allow us to represent events with different time granularities. Arcs are labeled with the disjunctive Allen primitives [1] between events. Table 1 illustrates the 13 Allen primitives.

After we translate the above story into the TCSP shown in Fig. 1, we may want to ask questions such as: is the story consistent? what are the possible arrival times of Mary? what is the earliest start time of John? check if a given scenario is consistent. To this end, a constraint propagation solving technique has been developed (at the numeric and symbolic levels) that is capable of returning the results in a very efficient time [16]. This technique is described below.

(1) *Numeric → symbolic conversion.* Perform the numeric → symbolic conversion on all the constraints. If one symbolic relation becomes empty then the constraint network is not consistent. The numeric → symbolic conversion works as follows: from the numeric information, we can extract the corresponding symbolic relation. An intersection of this relation with the given qualitative information will reduce the size of the latter which simplifies the size of the original problem. We use the following rules to extract the symbolic relations from the numeric ones. We assume here that e_i and e_j are two events, r_{ij} is the symbolic relation between them (initially set to the disjunction of the 13 Allen primitives), and inf_i, inf_j, sup_i, sup_j, d_i and d_j are respectively the earliest start time of e_i, earliest start time of e_j, latest end time of e_i, latest end time of e_j, duration of e_i and duration of e_j.
   (a) if $inf_i > sup_j$ then $r_{ij} \leftarrow$ Bi,
   (b) if $sup_i < inf_j$ then $r_{ij} \leftarrow$ B,
   (c) if $d_i < d_j$ then remove {E, Si, Fi, Di} from r_{ij},
   (d) if $d_i > d_j$ then remove {E, S, F, D} from r_{ij},
   (e) if $d_i = d_j$ then remove {D, Di, S, Si, F, Fi} from r_{ij},
   (f) if $inf_i + d_i > sup_j - d_j$ then remove {M, B} from r_{ij},
   (g) if $sup_i - d_i < inf_j + d_j$ then remove {Mi, Bi} from r_{ij},
   (h) if $inf_i > sup_j - d_j$ then remove {E, B, M, S, Si, O, Di} from r_{ij},
   (i) if $inf_i + d_i > sup_j$ then remove {E, B, M, F, Fi, D} from r_{ij},
   (j) if $sup_i < inf_j + d_j$ then remove {F, Fi} from r_{ij},
   (k) if $sup_i - d_i < inf_j$ then remove {S, Si, E} from r_{ij}.

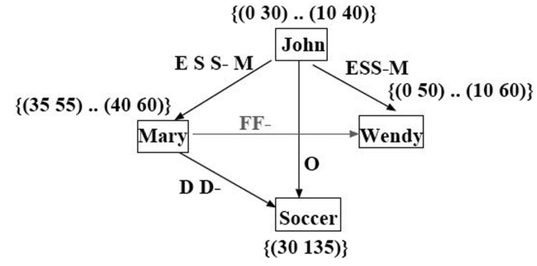

Fig. 1. A temporal constraint satisfaction problem (TCSP).

(2) *Local consistency.* Perform the path consistency algorithm PC-2 [21] (see Fig. 2) on the symbolic relations and the arc consistency algorithm AC-

Function PC()
// *INVERSE*: returns the inverse of a disjunctive relation
// Exp: $INVERSE(PF^{\smile}OD^{\smile}) = P^{\smile}FO^{\smile}D$
1. $PC \leftarrow false$
2. $L \leftarrow \{(i,j) | 1 \leqslant i < j \leqslant n\}$
3. **while** $(L \neq \emptyset)$ **do**
4. select and delete an (x, y) from L
5. **for** $k \leftarrow 1$ to n, $k \neq x$ and $k \neq y$ **do**
6. $t \leftarrow C_{xk} \bigcap C_{xy} \otimes C_{yk}$
7. **if** $(t \neq C_{xk})$ **then**
8. $C_{xk} \leftarrow t$
9. $C_{kx} \leftarrow INVERSE(t)$
10. $L \leftarrow L \cup \{(x,k)\}$
11. $t \leftarrow C_{ky} \cup C_{kx} \otimes C_{xy}$
12. **if** $(t \neq C_{ky})$ **then**
13. $C_{yk} \leftarrow INVERSE(t)$
14. $L \leftarrow L \cup \{(k,y)\}$

Fig. 2. Path consistency algorithm [21].

Function AC3()
// sopo: is an array of SOPOs
// R: set of disjunctive relations of the TCSP
1. $Q \leftarrow \{(i,j) \mid (i,j) \in R\}$
2. $AC \leftarrow true$
3. **While** $Q \neq Nil$ **Do**
4. $Q \leftarrow Q - \{(x,y)\}$
5. **If** $REVISE(x,y)$ **then**
6. **if** $Dom(x) \neq \emptyset$ **then**
7. $Q \leftarrow Q \sqcup \{(k,x) \mid (k,x) \in R \wedge k \neq y\}$
8. **else**
9. return $AC \leftarrow false$

Function REVISE(x, y)
// $compatible$: checks if two intervals are compatible
// regarding the symbolic relation they share
1. $REVISE \leftarrow false$
2. **For** each interval $a \in sopo[x]$ **Do**
3. **If** $\neg compatible(a,b)$ **for each interval** $b \in sopo[y]$ **Then**
4. remove a from $sopo[x]$
5. $REVISE \leftarrow true$

Fig. 3. Arc consistency algorithm for temporal constraints [16].

3 [12] that we have adapted in order to handle temporal constraints (see Fig. 3) on the temporal windows. If the resulting graph is not path or arc consistent then it is not consistent.

(3) *Backtrack search.* Perform a backtrack search algorithm [6] in order to look for a possible solution to the problem. Forward check through local consistency is used here during the backtrack search in order to prevent earlier later failure.

Figures 4–6 illustrate the resulting simplified constraint graph after applying the numeric $\rightarrow$ symbolic conversion, arc consistency and then path consistency to the problem of Example 1. When applying backtrack search (as described in step 3 above) to the simplified problem in Fig. 6, we obtain the solution depicted in Fig. 7. The goal here is to answer the following question: is the story consistent? This solution corresponds to the following consistent scenario satisfying all the

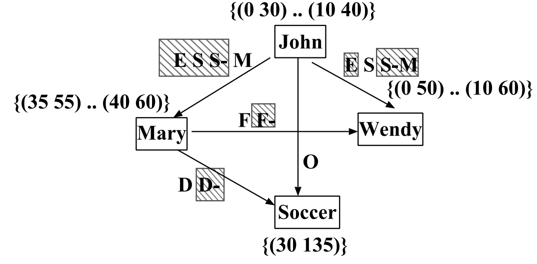

Fig. 4. Numeric $\rightarrow$ symbolic conversion applied to the problem in Fig. 1.

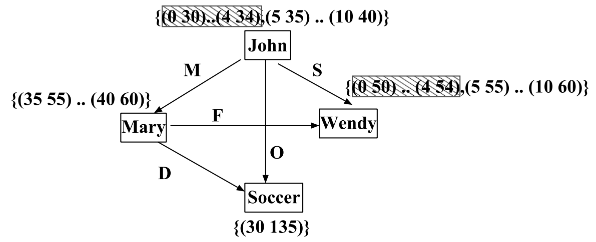

Fig. 5. Arc consistency applied to the problem in Fig. 4.

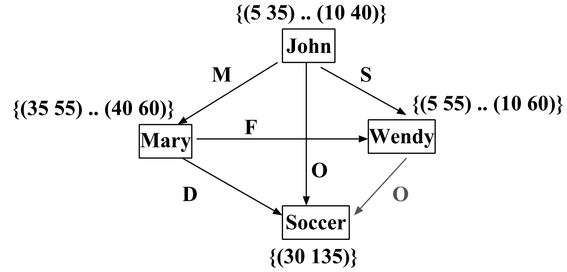

Fig. 6. Path consistency applied to the problem in Fig. 5.

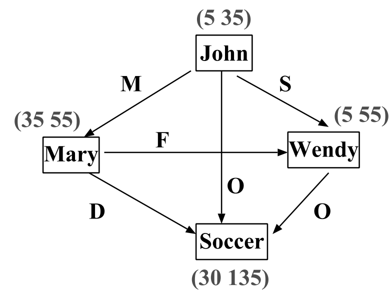

Fig. 7. Solution after applying backtrack search to the problem in Fig. 6.

–TCSP description and solution rules
context *TCSP* **inv**:
Name = –TCSP name;
EventNum = –event number; *ConstNum* = –constraint number;
TimeOut = –time deadline; *SolutionType* = –solution type;

– Temporal events and their domains
context *Events*[1] **inv**:
Name = –Events[1]'s name;
Domain = – Events[1]'s domain;
Duration = –Events[1]'s duration;
Step = –Events[1]'s step;
Value = –Events[1]'s interval;

•••

–Temporal constraints
context *Const*[1] **inv**:
Name = Const[1]'s name;
EventCons = –the two events involved in Const[1]
Relation = –Const[1]'s disjunctive Allen relation

•••

Fig. 8. TCSP template.

temporal constraints of the problem: *John arrived just as Mary started, at 7:35. John started just as Wendy started, at 7:05. Mary and Wendy arrived together, at 7:55, but started at different times. John's trip overlapped the soccer game. Mary's trip took place during the game. John's trip overlapped the soccer game. Wendy's trip overlapped the soccer game. The soccer game starts at 7:30 and lasts 105 minutes.*

In the real world, we often need to solve TCSPs in real time (within a given deadline). There are also situations where it is impossible or impractical to provide a complete and consistent solution for a given TCSP (also called over constrained TCSP). In order to address these issues, exact and approximation methods have been proposed and which are capable of trading search time for solution quality (number of solved constraints). Exact methods are based on Branch and Bound (BB) with constraint propagation [16]. Stochastic Local Search (SLS) [16], discrete Hopfield Neural Network (DHNN) [17] and Genetic Algorithms (GAs) [15] are examples of approximation methods. To assess the performances of each of these techniques, an empirical study of several TCSP instances has been conducted [15]. The results of this study favour SLS for under constrained problems, GAs for middle constrained problems, and both GAs and DHNN for over constrained problems. However, the exact method based on BB is the technique of choice in case we want to guarantee the quality of the returned solution.

As we mentioned in the introduction section, one of the features of our tool is to allow the user to change the description of the problem, by adding or removing some constraints in an incremental way, and check the effect of this change on the solutions. This process is handled by several dynamic solving algorithms proposed in [15,18].

context *TCSP* **inv**:
Name = "Soccer Game";
EventNum = 4; *ConstNum* = 5; *TimeOut* = 0; *SolutionType* = AllSolutions;

context Events*[1]* **inv**:
Name = "John";
Domain = [0, 40]; *Duration* = 30; *Step* = 1; *Value*= [0, 30];
context Events*[2]* **inv**:
Name = "Mary";
Domain = [35, 60]; *Duration* = 20; *Step* = 1; *Value* = [35, 55];
context Events*[3]* **inv**:
Name = "Wendy";
Domain = [0, 60]; *Duration* = 50; *Step* = 1; *Value* = [0, 50];
context Events*[4]* **inv**:
Name = "Soccer";
Domain = [30, 135]; *Duration* = 105; *Step* = 1; *Value* = [30, 135];

context *Const*[1] **inv**:
Name = "John and Mary"; *EventCons* = (1,2); *Relation* = [E, S, S−, M];
context *Const*[2] **inv**:
Name = "John and Wendy"; *EventCons* = (1,3); *Relation* = [E, S, S−, M];
context *Const*[3] **inv**:
Name = "John and Soccer"; *EventCons* = (1,4); *Relation* = [O];
context *Const*[4] **inv**:
Name = "Mary and Wendy"; *EventCons* = (2,3); *Relation* = [F, F−];
context *Const*[5] **inv**:
Name = "Mary and Soccer"; *EventCons* = (2,4); *Relation* = [D, D−];

Fig. 9. TCSP specification of Example 1.

3. A template for TCSP specification

In this section, we first introduce the extensions of OCL that are necessary to describe TCSP problems as well as the TCSP template based on the proposed extensions. Subsequently, we show how to instantiate the TCSP template to describe Example 1 presented in Section 2 and also a time scheduling problem. OCL, a formal specification language, is a subset of the UML standard. It is used to describe constraints of object-oriented systems that cannot be expressed by the diagrammatic notations [22]. Due to its formal semantics, OCL reduces the ambiguity of system descriptions and increases their precision [22]. In our work, to be able

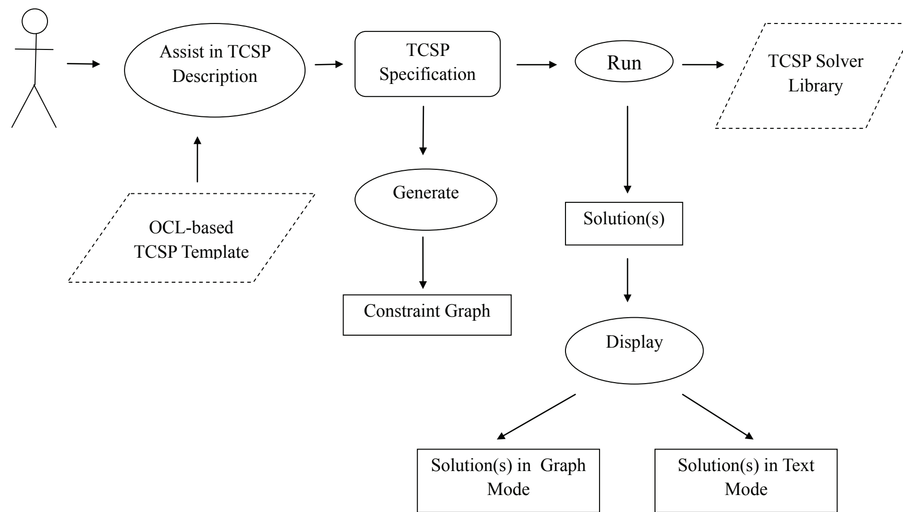

Fig. 10. TCSP specification and solving tool.

to specify temporal applications with the TCSP framework, we improve OCL with the following keywords.

TCSP description.

TCSP: a TCSP object.
Name: the name of a TCSP, an event or a constraint.
EventNum: the number of events within a TCSP.
ConstNum: the number of constraints within a TCSP.
Events: the list of temporal events.
Value: the initial value (interval) of an event.
Domain: the domain of an event denoted by [*start, end*] where *start* and *end* are respectively the earliest start and latest end times of the event.
Duration: the duration of an event.
Step: the discretization step.
Const: the list of temporal constraints.
EventCons: the two events involved in the temporal constraint.
Relation: the disjunctive Allen relation corresponding to the constraint.

Solution rules.

TimeOut: the allocated time in seconds needed to solve a given TCSP. The value 0 (default) means that there is no deadline.
SolutionType: the type of solutions to be returned; *AllSolutions* for all complete solutions and *OneSolution* for one single complete solution.

We may note that when TimeOut is equal to zero (corresponding to no time deadline), the constraint propagation method described in Section 2 is applied by default. Otherwise (TimeOut set to another value corresponding to a given deadline in seconds), the Branch and Bound (BB) with constraint propagation [16] is executed. In this latter case, the solver will either return a complete solution if it is found before the deadline or run until the timeout and returns the best solution obtained so far (the one satisfying most of the constraints by the given deadline). Note that in the case of timeout, only one solution is looking for. In both of these cases, the user has also the possibility to apply one of the other solving methods reported at the end of Section 2, such as Stochastic Local Search (SLS) [16], discrete Hopfield Neural Network (DHNN) [17] and Genetic Algorithms (GAs) [15] depending if the current problem is under, middle or over constrained.

Based on the OCL extensions, we present in Fig. 8 the TCSP template that models applications under temporal constraints throughout the TCSP framework. In OCL, first we need to declare the context (representing a class in the UML) for which we want to associate a number of constraints (called invariants). Here we define a context for three classes: a TCSP, an event and a constraint. In Fig. 9, we instantiate the TCSP template to produce the TCSP specification of Example 1 which has four events, five constraints and eight solutions.

4. TCSP specification and solving tool

Figure 10 illustrates the different functions of our TCSP specification and solving tool, including: assisting the user in the construction of the TCSP specifi-

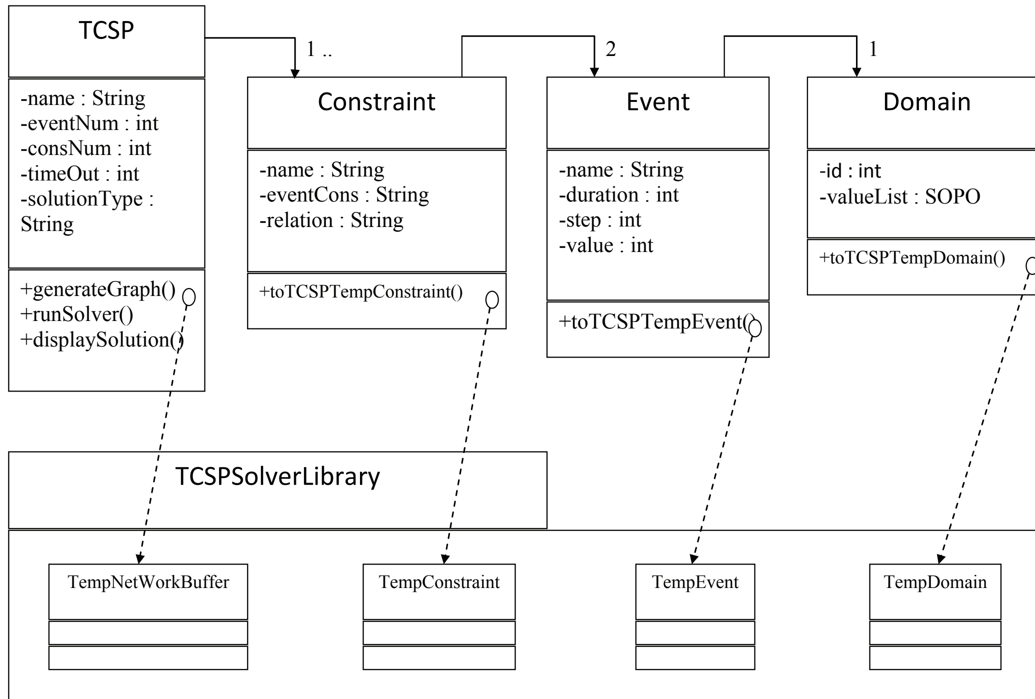

Fig. 11. Generic TCSP.

cation, generating the constraint graph directly from the TCSP specification in order to visualize the current temporal problem (as in Fig. 6), solving the TCSP problem according to the user's solution rules, and finally displaying the solution(s) in both text and graphic modes (as in Fig. 7). In Fig. 11, we design in an object-oriented approach our TCSP tool with a generic structure that models a wide variety of temporal constraint problems. Any TCSP application is defined with four classes: TCSP, Event, Domain and Constraint. The TCSP class is defined with three methods: generateGraph() to produce the constraint graph, runSolver() to call a certain constraint solver to produce the solution(s), and displaySolution() to view the solution(s). These four classes are then converted to their corresponding classes of the constraint class library being used. Furthermore, we model any existing constraint solver with the generic package TCSPSolverLibrary which comprises of four template classes.

Our tool has been fully implemented in Java SDK1.5 using the development platform Eclipse. Several Java libraries have been used, such as L2FProd.com to implement the TCSP description interface, and JGraphT (a free Java graph-theory library) that we have improved to draw the constraint graph and to display the corresponding solutions. The implementation of our tool relies on the Choco library [4] that we extended with the TCSP algorithms mentioned in Section 2.

Our TCSP tool first compiles the OCL-based TCSP specification into the Choco code and selects the appropriate algorithm (available in the extended Choco library). Even though the tool is experimented with Choco, it is developed in a way that it can be coupled with any of the existing constraint class libraries, such as ILOG [11], Prolog [8] and Java Cream [5].

Figure 12 depicts the application of our tool to Example 1. First the user enters the number of temporal events (here 4) and constraints (here 5) of the TCSP he wants to solve. The first version of the TCSP specification will then be generated from the TCSP template. Now the user completes it by providing the solution type (here allSolutions) as well as event and constraint information. From the resulting TCSP specification, the tool automatically produces the constraint graph and displays the solutions. For instance, in Fig. 12, the first found solution is shown in text and graphic modes. The button 'Next' is used to view the next solution. We may note that since there is no time deadline, the constraint propagation method given in Section 2 is applied by default.

The user can now modify the specification of the problem by adding or removing some constraints, or

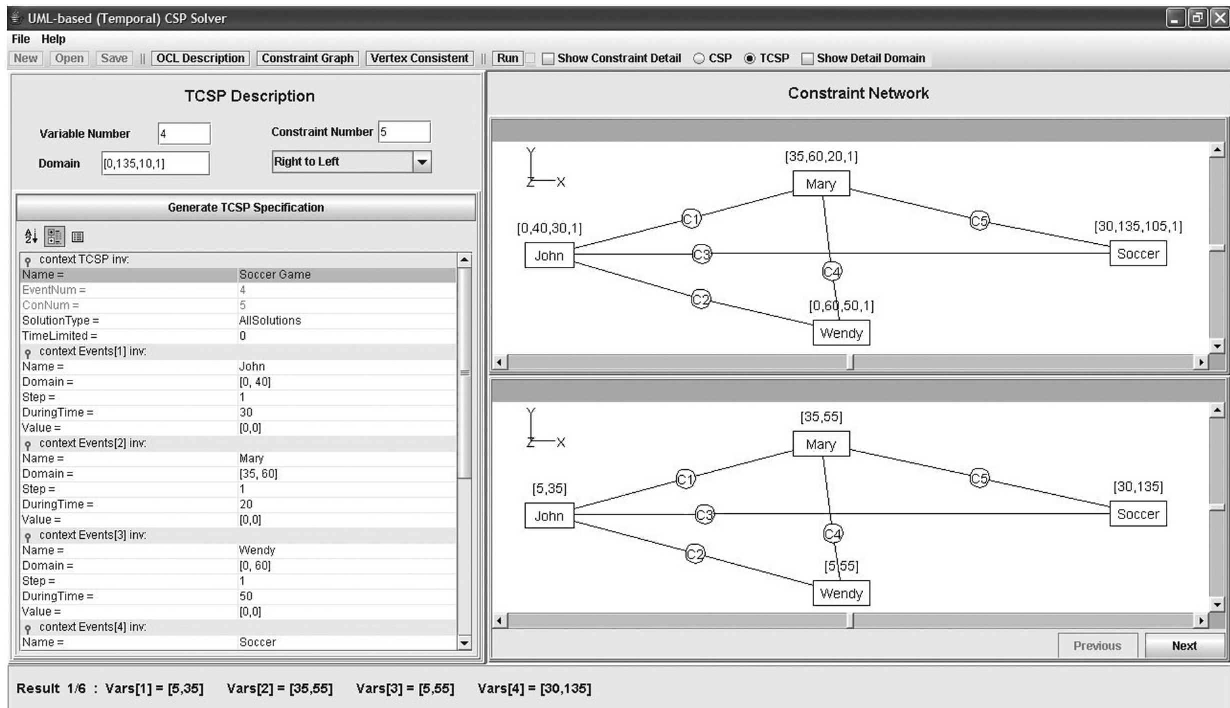

Fig. 12. Graphical user interface for Example 1.

changing the domains of the temporal events. The new constraint graph and solution(s) will then be regenerated automatically in an incremental manner.

5. Conclusion

In this paper, we focused on describing and solving numeric and symbolic temporal constraint problems through the TCSP framework. The tool comes with a friendly graphical user interface that enables the users to specify temporal problems. The solutions are automatically generated from the formal descriptions and users are not required to be knowledgeable about temporal solving techniques and constraint programming languages. In the near future, we are planning to improve the solving techniques so they can handle large problems in a reasonable response time. This can be done by studying different heuristics for variables and values ordering which will allow the backtrack search algorithm to find the solution sooner [19]. We will also tackle the case of over constrained problems where a complete solution satisfying all the constraints cannot be found. In this situation, we need to retract some of the constraints (those that are less important) in order to make the problem feasible. This can be done by using a dynamic variant of our solving techniques [15, 18]. Specifying complex and large temporal CSP problems as well as producing their solutions in text mode are both achievable with the proposed tool. Nevertheless, visualizing the constraint and solution graphs for large problems is not practical due to the large number of variables and constraints. One possible way to tackle this issue is to rely on distributed CSPs based on the agent technology [23]. In this case, a large and complex CSP will be partitioned into a set of subproblems that can be individually displayed in addition of being easier to solve. We intend to extend out TCSP tool based on the ideas of distributed CSPs.

References

[1] J. Allen, Maintaining knowledge about temporal intervals, *CACM* **26**(11) (1983), 832–843.

[2] B. Bauer, J.P. Muller and J. Odell, An extension of UML by protocols for multi-agent interaction, in: *the Fourth International Conference on MultiAgent Systems (ICMAS'00)*, IEEE, Washington DC (2000).

[3] C. Bettini, S. Mascetti and V. Pupillo, Gstp: A temporal reasoning system supporting multi-granularity temporal constraints, in: *IJCAI 2003* (2003), 1633–1634.

[4] Choco, http://www.emn.fr/z-info/choco-solver/, 2011.

[5] Cream, Class library for constraint programming in Java, http://bach.istc.kobe-u.ac.jp/cream, 2011.

[6] R. Dechter, Constraint processing, Morgan Kaufmann, 2003.
[7] R. Dechter, I. Meiri and J. Pearl, Temporal constraint networks, *Artificial Intelligence* **49** (1991), 61–95.
[8] D. Diaz and P. Codognet, Design and implementation of the GNU prolog system, *Journal of Functional and Logic Programming* **6**.
[9] R. Haralick and G. Elliott, Increasing tree search efficiency for constraint satisfaction problems, *Artificial Intelligence* **14** (1980), 263–313.
[10] P.V. Hentenryck, Constraint satisfaction in logic programming, The MIT Press, 1989.
[11] ILOG Solver, http://www.ilog.com/products/solver/, 2011.
[12] A.K. Mackworth, Consistency in networks of relations, *Artificial Intelligence* **8** (1977), 99–118.
[13] A.K. Mackworth and E. Freuder, The complexity of some polynomial network-consistency algorithms for constraint satisfaction problems, *Artificial Intelligence* **25** (1985), 65–74.
[14] M. Mouhoub, Reasoning with numeric and symbolic time information, *ICTAI 2000: Proceedings of the 12th IEEE International Conference on Tools with Artificial Intelligence*, IEEE Press (2000), 164–171.
[15] M. Mouhoub, Systematic versus non systematic techniques for solving temporal constraints in a dynamic environment, *Artificial Intelligence Communications*, IOS Press **17**(4) (2004), 201–212.
[16] M. Mouhoub, Reasoning with numeric and symbolic time information, *Journal of Artificial Intelligence Review*, Kluwer Academic Publishers **21**(1) (2004), 25–56.
[17] M. Mouhoub, A hopfield-type neural network based model for temporal constraints, *International Journal on Artificial Intelligence Tools* **13**(3) (2004), 533–546.
[18] M. Mouhoub, A new temporal CSP framework handling composite variables and activity constraints, *ICTAI 2005: Proceedings of the 17th IEEE International Conference on Tools with Artificial Intelligence*, IEEE Press (2005), 143–149.
[19] M. Mouhoub and B. Jafari, Heuristic techniques for variable and value ordering in CSPs, *GECCO 2011: Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, ACM (2011), 457–464.
[20] A. Trudel, Finding a single, all, or the most probable solution to a finite or non-finite interval algebra network, in: *Canadian Conference on AI*, Vol. 6085 of Lecture Notes in Computer Science, A. Farzindar and V. Keselj, eds, Springer, 2010, pp. 100–110.
[21] P. van Beek and D.W. Manchak, The design and experimental analysis of algorithms for temporal reasoning, *Journal of Artificial Intelligence Research* **4** (1996), 1–18.
[22] J. Warmer and A. Kleppe, The object constraint language 2.
[23] M. Yokoo and K. Hirayama, Algorithms for distributed constraint satisfaction: A review, *Journal of Autonomous Agents and Multi-agent Systems* **3** (2000), 185–207.