# TEMPLE - A Modeling Language for Resource Planning and Scheduling Problems

(1) Andreas Beer, (2) Johannes Gärtner, (3) Nysret Musliu, and (3) Werner Schafhauser

(1) Graz University of Technology, (2) Ximes Inc.,
(3) Vienna University of Technology

**Abstract.** In this paper we present TEMPLE, a language designed for modeling and solving resource planning and scheduling problems. In TEMPLE a particular planning or scheduling task is described with intervals, relationships between intervals, user-defined properties, curves and constraints. Once a problem has been defined in TEMPLE it is translated to the constraint based optimization language Comet where it is optimized by a local search algorithm. We demonstrate how a real-life scheduling problem can be modeled with TEMPLE and show that the solutions obtained with our approach are of satisfying quality.

**Key words:** domain specific language, resource planning and scheduling, constraint-based local search, Comet;

## 1 Introduction

In resource planning and scheduling problems one has to design shift plans which satisfy several requirements, such as staffing demands, labor rules, and other criteria specific for a particular industry. For instance, Figure 1 shows a typical input of a staff scheduling problem. As input we are given the staffing requirements over a planning period, depicted as red curve, specifying for each time point a minimum number of employees that should be working at that time.

The solution for this staff-scheduling problem provided in Figure 1 is a shift and break plan for five employees that does not violate the staffing requirement at all, since the curve of working employees corresponding to that solution (blue-line) does not fall below the requirement curve at any time point.

Obtaining good or close to optimal solutions to resource planning and scheduling problems improves the working conditions for employees and helps companies to deploy their staff efficiently and cost-savingly. Unfortunately, due to their discrete nature and an exponential number of possible solutions, many resource and staff scheduling problems are NP-hard, and consequently, they cannot be solved to optimality in polynomial time. Local search methods, such as tabu-search, variable neighborhood search or (stochastic) hill-climbing algorithms represent one possibility to obtain solutions for resource planning and scheduling problems of acceptable quality in reasonable time.

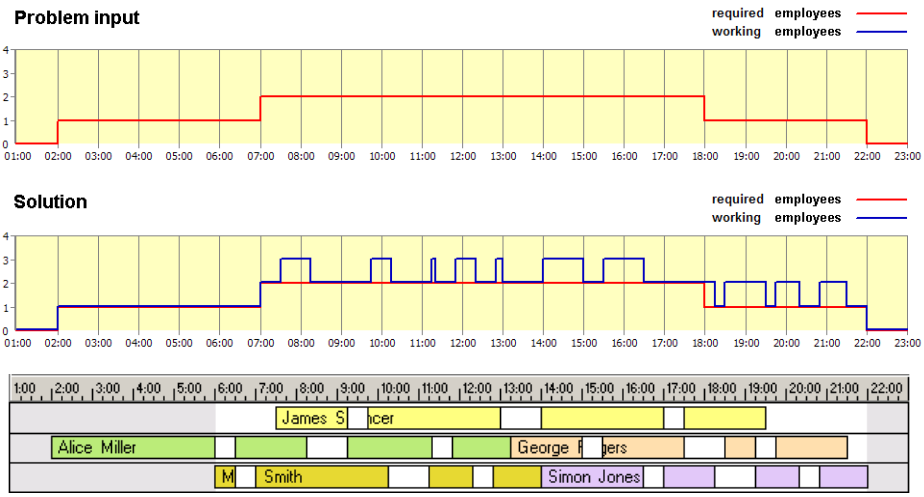


**Fig. 1.** Staffing requirements and a solution of a typical staff scheduling problem. The red curve represents the required staff over a twenty-hour planning period. The solution consists of a shift and break plan for the scheduled employees. In the given plan shifts are represented as colored vertical bars. The breaks for each employee are depicted as white blocks.

When designing and implementing a local search algorithm huge efforts are made to derive a suitable representation of the addressed problem which allows both an efficient evaluation of a solution's local neighbourhood and the quick application of changes to a current solution. Due to their complex problem structure, often involving many constraints on very specific problem properties, this is particularly true for resource and scheduling problems. An existing representation for a specific resource planning and scheduling problem cannot be applied to new problem arising in a different working area, without making comprehensive mollifications within the the algorithm's source code. Often, it is even necessary to implement a new algorithm from the scratch to model a newly emerged resource planning and scheduling task with great similarities to an already known and solved problem.

To overcome these drawbacks we developed the domain specific language TEMPLE. TEMPLE is aimed at supporting professional planners in modeling resource planning and scheduling problems for local search algorithms, by providing language elements which correspond to some of the common characteristics of many resource planning and scheduling problems. TEMPLE is developed within a research project in cooperation between Graz University of Technology, Vienna University of Technology and the consulting company Ximes Inc., which currently started to apply TEMPLE for the resource planning and scheduling problem emerging in its daily-business.

At this point we want to state the main contributions of this article:

- We propose the domain specific language TEMPLE. TEMPLE offers basic building blocks, common for many resource planning and scheduling problems, namely intervals, relations between intervals and properties, curves and constraints associated with a solution. In TEMPLE properties, curves and constraints are stepwise derived from already defined or existing ones by using the operational syntax of the constraint-based optimization language Comet [7]. That way TEMPLE supports a planner to model a problem in a very natural and modular fashion while requiring only basic programming skills.
- We demonstrate how to model a complex real-world break-scheduling problem for supervision personnel [1] in TEMPLE. The effort for deriving a TEMPLE model for the considered break-scheduling problem can be quantified with one to a few man days, whereas to obtain the problem model for the algorithm described in [1] several man-weeks had to be spent.
- We present a TEMPLE-Comet compiler translating a problem model written in TEMPLE into source code of the constraint-based optimization engine Comet.
- We use this compiler to translate the problem model to the Comet optimization language. In addition we implement a simple hill-climbing algorithm for the translated comet model and demonstrate that the translated model can still be optimized by a local search technique in reasonable time. For that purpose, we perform experiments on ten real-life and five randomly generated benchmark instances for the break-scheduling problem for supervision personnel. Our computational results reveal that the hill-climbing algorithm is able to improve the quality of the initial solutions significantly and the obtained solution shift plans are of sufficient quality to be applied in practice.

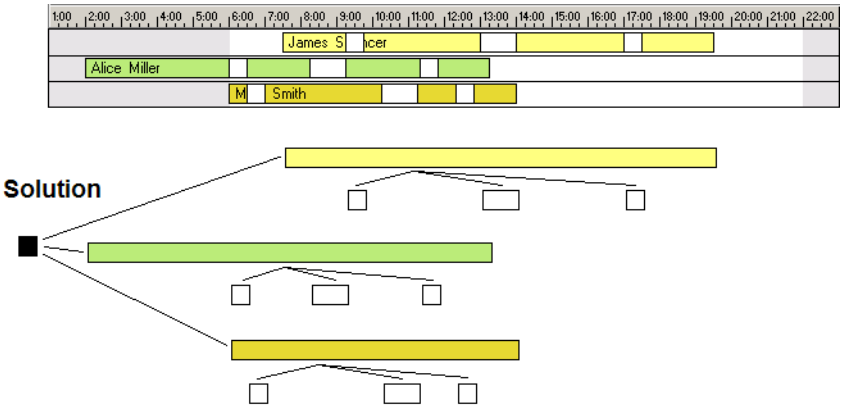


**Fig. 2.** Interval relations in a typical shift plan. The entire solutions contains several shifts, and each shift may be associated with the breaks scheduled within it and vice versa.

## 2 The Modeling Language TEMPLE

Before introducing the domain specific language TEMPLE we present several elements, common to many resource planning and scheduling problems:

**Intervals** Shifts, breaks, meetings, tasks, processes and even the entire problem itself can be considered as time intervals. A time interval is characterized by three basic properties, its start, its end and its duration.

**Relations between intervals** In a resource planning and scheduling problem there exists several relations between intervals. For instance, Figure 2 shows the relations which may be identified within a shift plan containing breaks. The entire solutions contains several shifts, shifts in turn contain several breaks.

**Derived Properties** For an interval a new property can be derived depending on already defined properties of the interval itself and the intervals in relation to the interval. For instance, the net-duration of a shift can be derived by subtracting the duration of its associated breaks from its original duration.

**Derived Curves** Curves can be derived from intervals. For instance, in Figure 1 the curve of available employees is derived from the shifts and breaks of the underlying shift plan. Moreover, curves can also be derived from other curves, e.g., by summing up two curves we will obtain again new curve. Note, that also property values can be derived from curves, e.g., we may assign the sum of all curve values to a property value.

**Constraints** Every resource planning and scheduling problem imposes several constraints on a possible solution. Constraints can be regarded as a special case of a property, where instead of a property value the violation degree of a constraint is computed.

The domain specific language TEMPLE consists of the following language elements representing the common characteristics of resource planning and scheduling problems mentioned above: interval declarations, relations declarations, property definitions, curve definitions, hard constraint definitions, and soft constraint definitions.

### 2.1 Interval Declarations

A TEMPLE-model starts with one or several interval declarations. A planner must specify the following statement in order to declare that time intervals representing shifts are part of a problem model:

```
Interval Shift;
```

Each interval declared in the problem model has four basic properties: `Start`, `End`, `Duration` and `Active`. These basic properties can be used to define additional derived properties of intervals in the addressed resource planning and scheduling problem. The basic property `Active` is a boolean property which can take only the values zero (=inactive) or one (=active). This property was introduced in order to activate or deactivate intervals within a solution of a specific problem model.

### 2.2 Relation Declarations

An interval relation declaration associates two kinds of intervals with each other. The interval relation declaration

```
Shift <-> Break;
```

can be used to model the fact that each shift contains several breaks and each break is in relation with the shift in which it is scheduled.

### 2.3 Property Definitions

Beside the basic interval properties `Start`, `End`, `Duration` and `Active`, we may define additional properties for an interval which are computed from already existing properties of an interval itself or from related intervals. In the definition for an additional property a planner specifies the type of interval a property is associated with, the name of the additional property and he/she lists the related intervals from which the additional property is derived. Furthermore the planner provides a piece of code in which he/she describes how the value of the new property is computed. This code has the same syntax as the constraint optimization language Comet. For instance, the net-duration of a shift containing several breaks can be computed from the duration of a shift and the duration of its associated breaks as follows:

```
Property Shift::NetDuration(Shift thisShift, Shift.Break[] breakInShift)
{
   int shiftDuration = thisShift.Duration;
   int breakTime     = sum(i in breakInShift.getRange()) (breakInShift[i].Duration)

   value = shiftDuration - breakTime;
}
```

### 2.4 Curve Definitions

Curves can be derived from properties and other curves. To describe the way a curve is computed from already existing intervals and curves, the planner accesses the curve to be computed with the key word `curve`. The planner is provided several methods, in order to specify how properties increase or decrease the curve value along a specific period of time, and there exist also methods for adding and subtracting already existing curves from the curve specified within the problem definition. The following curve definition specifies how the curve of working employees within a solution is computed from all its shifts. Along the duration of each shift the curve of working employees is incremented by one.

```
Curve Solution::WorkingEmployees(Solution.Shift[] associatedShifts)
{
   forall(i in associatedShifts.getRange())
      curve.pulse(associatedShifts[i].Start, associatedShifts[i].End, associatedShifts[i].Active);
}
```

### 2.5 Hard and Soft Constraint Definitions

Hard and soft constraints are defined similarly as additional properties of time intervals. The only difference is that for constraints we define the violation degree of a constraint instead of a property value. The following definition of a hard constraint requires that each shift contains at least a certain amount of break time. If the constraint is violated the difference to the required break time acts as the violation degree of the hard constraint.

```
HardConstraint Shift::MinimumBreakTime(Shift thisShift, Shift.Break[] breakInShift)
{
   int total_break_time = sum(i in _breakInShift.getRange()) (_breakInShift[i].Duration);

   violation = max (minimum_break_time - total_break_time, 0);
}
```

Soft constraints are defined in almost the same manner as hard constraints. Additionally, a soft constraint may be associated a weight representing the its significance in a problem's objective function.

```
SoftConstraint Break::MinDuration(Break thisBreak) weight(10)
{
   violation = max (minimum_break_duration - thisBreak.Duration, 0);
}
```

## 3 The Break Scheduling Problem for Supervision Personnel

To demonstrate how problems can be successfully modeled and solved in TEMPLE, we show how a real-life break scheduling problem arising in the area of supervision personnel [1] can be formulated within our domain specific language. In the break scheduling problem for supervision personnel [1] we are concerned with shift plans in which each shift must contain a certain amount of break time. The breaks must be scheduled in such a way that the resulting plan satisfies several constraints modeling labor rules and legal requirements and minimizes shortage and excess of working employees according to given staffing requirements. Formally, as input for the break scheduling problem we are given:

- A planning period formed by $T$ consecutive time slots $[a_1, a_2), [a_2, a_3), ..., [a_T, a_{T+1}]$ all having the same slot length of, e.g., five minutes. The time points $a_1$ and $a_{T+1}$ represent the beginning and end of the planning period.
- $n$ *shifts* $s_1, s_2, ..., s_n$ representing employees working within the planning period. Each shift $s_i$ has an adjoined parameter $s_i.breaktime$ specifying the required amount of break time for $s_i$ in time slots.
- The *staffing requirements* for the planning period. For each time slot $[a_t, a_{t+1})$ we are given an integer value $r_t$ indicating the required number of employees that should be working during time slot $[a_t, a_{t+1})$. An employee is considered to be working during time slot $[a_t, a_{t+1})$ if neither he/she has a break during time slot $[a_t, a_{t+1})$ nor his/her break has ended at time point at. After a

break an employee needs a full time slot, usually five minutes, to reacquaint him- or herself with the altered situation. Thus also during the first time slot following a break an employee is not considered to be working.

Given a planning period, a set of shifts the associated total break times, and the staffing requirements, a solution to the break scheduling problem is a shift plan containing breaks. Among all shift plans we aim at finding an optimal one that satisfies several hard constraints and minimizes the weighted sum of violation degrees of two soft constraints on the excess and shortage of working employees. In the following we give a short verbal description of the involved hard and soft constraints. For the formal problem description see [1]:

$H_1$ **Break Positions:** Each break $b_j$ may start at the earliest a certain number of time slots after the beginning of its associated shift $s_i$ and may end at the latest a given number of time slots before the end of its shift.

$H_2$ **Lunch Breaks:** A shift $s_i$ can have several lunch breaks, each of which is required to last a specified number of time slots and should be located within a certain time window after the shift start.

$H_3$ **Duration of Work Periods:** Breaks divide a shift into several work and rest periods. The duration of work periods within a shift must range between a required minimum and a maximum duration.

$H_4$ **Minimum Break Times after Work Periods:** If the duration of a work period exceeds a certain limit the break following that period must last a given minimum number of time slots (min. ts count).

$H_5$ **Break Durations:** The duration of each break $b_j$ must lie within a specified minimum and maximum value.

$S_1$ **No Shortage:** In each time slot $[a_t, a_{t+1})$ at least $r_t$ employees should be working ($r_t$ are the staffing requirements in time slot $t$).

$S_2$ **No Excess:** In each time slot $[a_t, a_{t+1})$ at most $r_t$ employees should be working ($r_t$ are the staffing requirements in time slot $t$).

Our goal is to obtain a shift plan with breaks which satisfies all hard constraints $H_1 - H_9$ and minimizes the following weighted sum of soft constraint violation degrees: $2 \cdot S_1 + 10 \cdot S_2$.

We modified the original problem description of the break scheduling problem for supervision personnel by changing five constraints in [1], $H_1 - H_5$, from soft constraints to hard constraints. We made these changes in the problem model in order to provide also an example on how hard constraints can be modeled in TEMPLE.

# 4 The TEMPLE-Model of the Break-scheduling Problem for Supervision Personnel

## 4.1 Intervals

Obviously, a solution for the break-scheduling problem consists of *shifts* and *breaks*. In addition, also the staffing requirements can be regarded as intervals. For instance, if there are three employees required to be working in time slot $[a_t, a_{t+1}]$, we can represent that demand with three intervals, each starting at time point $a_t$ and ending at $a_{t+1}$. Moreover, we also declare an interval representing the entire solution, which is associated the properties, curves, and constraints, concerning the entire shift plan (solution), such as shortage and excess of working employees.

```
Interval Solution;
Interval Requirement;
Interval Shift;
Interval Break;
```

## 4.2 Interval Relations

Considering the problem description of the addressed break-scheduling problem we can observe several relations between the declared intervals. Breaks are scheduled within shifts. Our entire solution consists of all shifts given by the input and also by the intervals representing the staffing requirements. These considerations lead to the following declarations of interval relations within our TEMPLE-model.

```
Shift       <->  Break;
Solution    <->  Shift;
Solution    <->  Requirement;
```

## 4.3 Properties, Curves and Constraints

Due to space limitations we cannot present all the definitions of properties, curves and constraint of the break-scheduling problem for supervision personnel in this article. Instead we give the definition of hard constraint $H_1$ Break Positions and soft constraint $S_2$ No Excess and the definitions of all properties and curves which are necessary to formulate these constraints.

The hard constraint $H_1$ Break Position requires that each break $b_j$ starts at the earliest a certain number of time slots after the beginning of its associated shift $s_i$ and ends at the latest a given number of time slots before the end of its associated shift. To model this constraint we define two new properties of breaks representing a break's distance to the start of its associated shift and its distance to the end of the associated shift.

```
Property Break::DistanceToShiftStart (Break thisBreak, Break.Shift[] associatedShift)
{
   value = thisBreak.Start - associatedShift[1].Start;
}
```

```
Property Break::DistanceToShiftEnd (Break thisBreak, Break.Shift[] associatedShift)
{
   value = associatedShift[1].End - thisBreak.End;
}
```

In our problem instance, a break violates the hard constraint BreakPosition, if it is scheduled within the first or last thirty minutes of a shift. The violation degree of constraint BreakPosition is the number of time slots that lie within the first or last `minimum_distance_to_shift_border` time slots of a each shift. By the the help of the derived properties *DistanceToShiftStart* and *DistanceToShiftEnd* we are now able to express this violation degree and consequently to model the hard constraint Break Position.

```
HardConstraint Break::BreakPosition (Break thisBreak)
{
   int distanceToShiftStart =
   (max(minimum_distance_to_shift_border - thisBreak.DistanceToShiftStart, 0)) * thisBreak.Active;

   int distanceToShiftEnd   =
   (max(minimum_distance_to_shift_border - thisBreak.DistanceToShiftEnd  , 0)) * thisBreak.Active;

   violation = distanceToShiftStart + distanceToShiftEnd;
}
```

To compute the excess of working employees we first derive the a curve representing the staffing requirements from the requirement intervals in our problem model.

```
Curve Solution::StaffingRequirements(Solution.Requirement[] requirement)
{
   forall(i in requirement.getRange())
      curve.pulse(requirement[i].Start, requirement[i].End, requirement[i].Active);
}
```

Then, for each shift we define a curve representing the shift's working time pattern. The curve takes a value of one along the duration of the shift. Breaks decrease the value of a curve by one. Also, during the time slot after the end of a curve an employee is not considered to be working, because he/she has to reacquaint him- or herself an altered situation. Thus, also in each time slot succeeding a break the curve value is decremented.

```
Curve Shift::WorkingTimePattern(Shift thisShift, Shift.Break[] breakInShift)
{
   //a shift increases the working time pattern by 1
   curve.pulse(thisShift.Start, thisShift.End, thisShift.Active);

   forall(i in breakInShift.getRange())
   {
      //a break decreases the working time patter by 1
      curve.pulse(breakInShift[i].Start, breakInShift[i].End, breakInShift[i].Active, -1);

      //one time slot after break an employee is still not considered to working
      curve.pulse(breakInShift[i].End+1, breakInShift[i].End+1, breakInShift[i].Active, -1);
   }
}
```

By summing up the working time patterns of all shifts within a solution we obtain the curve of working employees within that solution. Then, by subtracting

the staffing requirement curve we obtain a curve representing the deviation from the staffing requirements. A positive curve value at a specific time slot indicates excess of working employees whereas negative curve values show that shortage of employees occurs at that time slot.

```
Curve Solution::DeviationFromRequirements(Solution thisSolution, Solution.Shift[] shiftInSolution)
{
   forall(i in shiftInSolution.getRange())
      curve.add(shiftInSolution[i].WorkingTimePattern);

      curve.subtract(thisSolution.StaffingRequirements);
}
```

By extracting the positive curve values we obtain a curve representing the excess of working employees within a solution to the breaks scheduling problem.

```
Curve Solution::ExcessCurve(Solution thisSolution)
{
   curve.addPositiveValues(thisSolution.DeviationFromRequirements);
}
```

Now we sum up all entries within of a solution's excess curve and obtain a property representing the excess of working employees associated with a solution of the break scheduling problem for supervision personnel.

```
Property Solution::Excess(Solution thisSolution)
{
   Curve excessCurve = thisSolution.ExcessCurve;

   value = sum(position in excessCurve.period())     (excessCurve.value(position));
}
```

This number is also the violation degree of the soft constraint $S_2$ No Excess. According to the objective function of the break scheduling problem we weight the violation degree of constraint $S_2$ No Excess with a weight of two.

```
SoftConstraint Solution::NoExcess(Solution thisSolution) weight(2)
{
  violation = thisSolution.Excess;
}
```

The entire TEMPLE model of the break scheduling problem for supervision personnel contains eleven property definitions, six curve definitions, five hard constraint and two soft constraint definitions. In total all constraints of the break scheduling problem were modeled by using only 325 lines of source code, whereas problem model for the min-conflicts based local search algorithm described in [1] comprised 6014 lines of code.

Moreover, the above example definitions illustrate that by defining the properties, curves and constraints of a specific resource planning and scheduling problem stepwise, a planner is forced to write very well structured code. The resulting problem model is clearly arranged and consequently easy to modify, extend and maintain.

Concerning development time, we experienced that to the TEMPLE model for the break-scheduling problem for supervision personnel model one man to few days was needed, whereas to obtain the problem model for the algorithm described in [1] several man-weeks had to be spent.

# 5 Optimizing the TEMPLE-model with Comet

## 5.1 The Temple-Comet Compiler

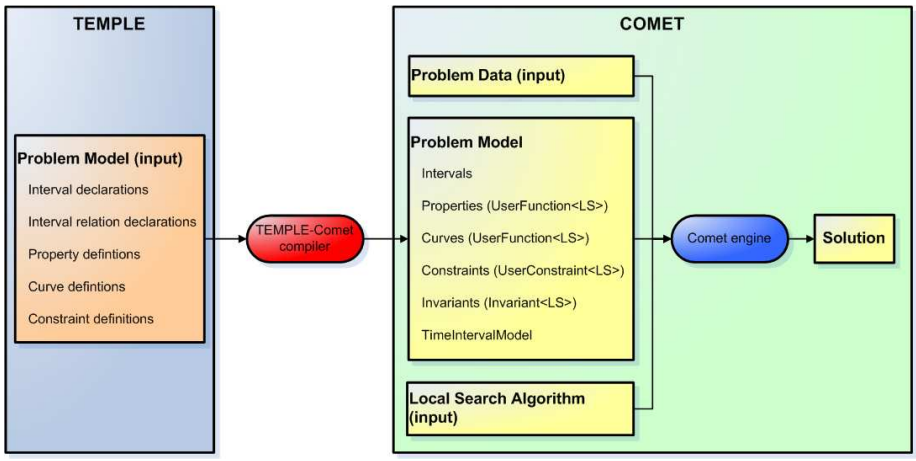


**Fig. 3.** The TEMPLE problem model of the resource planning and scheduling problem is translated into classes of the constraint-based optimization language Comet. Together with the input data for a problem instance and the source code of a local search algorithm the generated classes are processed by the Comet local search engine.

We developed a TEMPLE-Comet compiler to translate the TEMPLE model of a specific resource planning and scheduling problem into the programming language Comet. Comet is an object-oriented language featuring a constraint-based local search engine. The key idea behind local search techniques is to repeatedly apply small changes to intermediate solutions in order to find solutions with higher quality. In each step, local search techniques examine solutions closely related to the current one, the so-called local neighborhood, and select one solution within that local neighborhood to be the next current solution. Usually, the local neighborhood is computed by applying small changes to the current solution. In terms of local search techniques these small changes are called moves.

Our compiler generates for each interval declaration a class containing class member variables as well as set- and get-methods for related intervals, properties, curves and constraints. Moreover for definitions of properties, curves and constraints the compiler generates classes extending Comet's user defined functions (`UserFunction<LS>`) and constraints (`UserConstraint<LS>`).

While parsing the problem written in TEMPLE, our compiler identifies occurrences of properties and curves in the definition of derived properties, curves and constraints. The source code specified within the property, and constraint definitions is slightly modified and inserted into user defined functions and constraints. In a local search algorithm the modified code is used to determine the

new value or violation degree a property or constraint will take if a certain move is carried out. For curves the modified code calculates which positions of the curve will change and which values the curve will take at these positions under a certain move. In this way we guarantee that changes of curves are determined efficiently.

Once a particular move has been selected and carried out, we must propagate the changes resulting from the chosen move to obtain the current solution of our local search algorithm. For that purpose our compiler builds a class extending Comet's user defined invariant `Invariant<LS>` for each definition of a property, curve and constraint, alters the source code specified within a definition slightly, and inserts it into the invariant. Whenever a move is carried out and a property, curve or constraint is affected by that move, the embedded code is executed and the corresponding property, curve or constraint is recomputed again.

Finally, our compiler also generates a class `TimeIntervalModel` which is responsible for the instantiating all created classes in the correct order and for posting hard and soft constraints into Comet constraint systems (`ConstraintSystem<LS>`). The time interval model is also used by user defined local search algorithms to retrieve the hard and soft constraint violation degrees of a solution and to evaluate the impact of potential moves to the current solution.

Figure 3 illustrates the transforming steps and actions performed by the TEMPLE-Comet compiler and indicates the input required to be given by a planner in order to formulate and solve a resource planning and scheduling problem. Beside a the TEMPLE model of a problem a planner must also provide the problem data for the problem instance, including also an initial solution and a local search algorithm for Comet model produced by the compiler.

### 5.2 Initial Solution and Problem Optimization

For an initial solution for the break-scheduling problem we want to obtain breaks satisfying all hard constraints of the break scheduling problem. The problem of finding such a solution can be formulated as small temporal constraint satisfaction problem (STP) which can be solved in polynomial running time [2]. The STP formulation for obtaining a solution for the break scheduling problem without any hard constraint violations can be found in [1].

Next we successively improve the objective function value of that initial solution by applying a simple hill-climbing heuristic. In each iteration of the heuristic we randomly select a break and evaluate the neighbourhood solutions obtained by assigning this break a new position or by swapping it with another break in its associated shift. We consider those neighbourhood solutions not violating any hard constraints and having a smaller objective function value than the current solution. Out of this neighbourhood we select the solution with the best objective function value to be the next solution in our search heuristic.

## 6 Computational Results

Up to this point it remains unclear whether the problem model created by the TEMPLE-Comet compiler can be used to optimize solutions of the considered break-scheduling problem. It could be the case that the transformation from the TEMPLE model into Comet code inserts sources of inefficiency and thus, the obtained comet model is not suited to constraint based local search algorithms.

To evaluate the problem model generated by our compiler we applied it and the simple hill-climbing heuristics to ten real-life benchmark instances, and five randomly generated instances which are publically available under `http://www.dbai.tuwien.ac.at/proj/SoftNet/Supervision/Benchmarks/`. For each instance the time slot length is five minutes and the planning period comprises an entire week. We ran the hill-climbing algorithm ten times for each instance on a QuadCore Intel Xeon 5345 with 48 GB RAM. A single test run was executed with a one-hour time limit.

Table 1 shows for each instance the number of shifts and scheduled breaks and reports the best and mean objective function value obtained in ten and the corresponding standard deviation. Column "Mean Initial" gives the mean objective value of the initial solutions at the beginning of each run.

When comparing the mean and best results with the mean initial objective function value, we see that the function value was improved significantly by the hill-climbing algorithm. Figure 4 presents parts of the requirement curves (red curve) and the curves of working employees (blue curve) in the best solutions obtained for the real-life benchmark instance 3si2ji2. The presented detail of solution 3si2ji2 does not contain any periods with shortage of employees at all. For the considered instance, excess of employees cannot be avoided due to the characteristics of the input shift plan. For the considered curves, excess of employees cannot be avoided due to the characteristics of the input shift plan. We conclude that with the obtained problem model we are able to compute solutions of acceptable quality in reasonable time, even with a simple hill climbing heuristic.

| Instance | Shifts | Breaks | Mean Initial | Best | Mean | Std. Dev. |
|---|---|---|---|---|---|---|
| 2fc04a | 135 | 1113 | 13658.0 | 4760 | 4920.8 | 93.60 |
| 2fc04b | 126 | 1064 | 14271.6 | 4248 | 4470.0 | 67.20 |
| 3fc04a | 124 | 1048 | 13988.4 | 3978 | 4197.6 | 104.40 |
| 3si2ji2 | 151 | 1182 | 10948.0 | 4432 | 4197.6 | 104.40 |
| 4fc04a | 124 | 1050 | 13825.2 | 4038 | 4170.0 | 81.60 |
| 4fc04b | 125 | 1048 | 13484.8 | 3304 | 3535.6 | 113.28 |
| 50fc04a | 130 | 1091 | 14898.4 | 4534 | 4712.8 | 66.48 |
| 50fc04b | 126 | 1069 | 15274.8 | 5334 | 5449.2 | 99.84 |
| 51fc04a | 129 | 1081 | 14960.0 | 4784 | 5036.0 | 100.80 |
| 51fc04b | 126 | 1065 | 15563.2 | 5914 | 6078.4 | 83.76 |
| random1-1 | 137 | 962 | 10136.4 | 1116 | 1405.2 | 152.40 |
| random1-5 | 141 | 950 | 12172.8 | 1860 | 2034.0 | 128.40 |
| random1-7 | 157 | 1089 | 11629.2 | 1242 | 1514.4 | 81.12 |
| random2-1 | 179 | 1255 | 14802.8 | 2606 | 2776.4 | 96.00 |
| random2-4 | 162 | 1075 | 12933.2 | 2528 | 2728.4 | 69.84 |

**Table 1.** Test results for real-life and randomly generated benchmarks

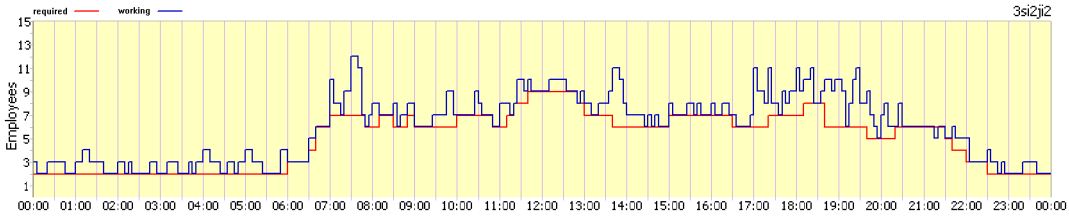


**Fig. 4.** Staffing requirements and curve of working employees for a part of the best solution obtained for the real-life benchmark instance 3si2ji2.

## 7 Related Modeling Languages

Several languages have been developed for modeling combinatorial optimization problems. This includes OPL [6], COMET [7], ESRA [3], ESSENCE [4], ZINC [8], ASPEN [5] etc. Due to space limitation we will discuss here shortly only some of the existing languages, and the reader is referred to [4] and ZINC [8] for a more comprehsive overview.

OPL [6] language is used to formulate mathematical programming and combinatorial optimization problems. OPL gives support for constraint programming, and a specialized support for scheduling and resource allocation applications. COMET [7] programming language provides an expressive constraint language to model combinatorial optimization problems. Moreover, COMET also offers a rich search language that abstracts different components of local search algorithms. ESRA [3] is a relational constraint language for modelling combinatorial problems. It supports finite domains and can compute efficient models for lower-level constraint programming languages. This language uses mathematical and logical notation to specify the problem. ESSENCE [4] is a formal language based on concepts of discrete mathematics and it is also used to specify combinatorial problems. In this language the domain of decision variables consist of combinatorial objects, and the language also provides the constraints that operate on such variables. This enables problems to be specified naturally and the language provides a high level of abstraction. Zinc [8] uses also natural mathematic-like notation to specify models for combinatorial problems. This language allows the user to extend it to new application domains by adding new predicates and functions, and it also includes continous variables. ASPEN [5] is a system that provides a constraint modeling language that is used to define problems in spacecraft operations domain. Furthermore, the system provides several search strategies to solve planning/scheduling problems. The main data structure is an activity, and the system provides the possibility to maintain hierarchical relationships between activities. Components for expressing and maintaining the temporal constraints are also provided. ASPEN has been used for several problems in NASA.

## 8 Conclusions and Future Work

In this article we presented TEMPLE a language designed for modeling resource planning and scheduling problems. In TEMPLE a planner can define a specific problem by using language elements for common features of resource planning and scheduling problems: intervals, interval relations and properties, curves and constraints associated with a solution of the addressed problem. To demonstrate TEMPLE's expressiveness we used TEMPLE to model a complex real-world break-scheduling problem arising from the area of supervision personnel. We observed that this problem could be modeled very quickly (within a few days) and the obtained problem model was very well structured and therefore easy to change, extend or maintain. Moreover we presented a compiler translating the model obtained with TEMPLE to the constraint language Comet and solved the problem with a simple hill-climbing heurstic. Experimental results on real-life and random benchmarks revealed that our approach is able to compute solutions of acceptable quality within a reasonable amount of time. For future work we plan to develop a domain specific language for representing an initial solution and a problem specific local search algorithm.

## 9 Acknowledgments

We thank Pascal van Hentenryck, Laurent Michel and the support team at Dynadec Corp. for their immediate answers of our request concerning Comet.

## References

1. A. Beer, J. Gärtner, N. Musliu, W. Schafhauser, and W. Slany. A break scheduling system for supervision personnel using ai techniques. *Accepted for publication in IEEE Intelligent Systems*.
2. R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Artif. Intell.*, 49(1-3):61–95, 1991.
3. P. Flener, J. Pearson, and M. Ågren. Introducing esra, a relational language for modelling combinatorial problems. In *CP*, page 971, 2003.
4. A. M. Frisch, W. Harvey, C. Jefferson, B. M. Hernández, and I. Miguel. Essence : A constraint language for specifying combinatorial problems. *Constraints*, 13(3):268–306, 2008.
5. A. S. Fukunaga, G. Rabideau, S. Chien, and D. Yan. Aspen: A framework for automated planning and scheduling of spacecraft control and operations. In *In Proc. International Symposium on AI, Robotics and Automation in Space*, 1997.
6. P. V. Hentenryck, I. Lustig, L. Michel, , and J.-F. Puget. *The OPL Optimization Programming Language*. The MIT Press, 1999.
7. P. V. Hentenryck and L. Michel. *Constraint-Based Local Search*. The MIT Press, Massachusetts, 2005.
8. K. Marriott, N. Nethercote, R. Rafeh, P. J. Stuckey, M. G. de la Banda, and M. Wallace. The design of the zinc modelling language. *Constraints*, 13(3):229–267, 2008.