Component Library of Problem Models for ALMM Solver

Sławomir Korzonek* and Ewa Dudek-Dyduch

*Advanced Technology Systems International, Zabierzów, Poland; bDepartment of Automatics and Biomedical Engineering, AGH University of Science and Technology, Kraków, Poland

ABSTRACT

The article presents the software tool named Algebraic-Logical Meta-Model of Multistage Decision Processes (ALMM) Solver to solve combinatorial and discrete optimization problems including non-deterministic polynomial-time hard ones. The solver utilizes the modelling paradigm named ALMM and its theory, both developed by Dudek-Dyduch E. ALMM enables a unified approach to creating discrete optimization problem models and representing knowledge about the problems as well as the solving methods for them. The models have a component structure and are stored in a module named Problem Model Library (PML). A new, extended modular structure of ALMM Solver is presented together with a basic layout of the PML. The article focuses on application of the component technology to the PML designing and component modelling of different type of limitations. General library source code organization is proposed. Considerations are illustrated by means of a sample class of scheduling problems.

ARTICLE HISTORY

Received 9 May 2017
Accepted 2 June 2017

KEYWORDS

ALMM Solver; optimizer; component technology; Algebraic-Logical Meta-Model (ALMM); multistage decision process; discrete optimization problems; scheduling problem; software tool; library of models

1. Introduction

The paper describes a software tool for solving combinatorial and discrete optimization problems named Algebraic-Logical Meta-Model of Multistage Decision Processes (ALMM) Solver. The first, initiative structure of ALMM Solver and the related requirements are presented in Dudek-Dyduch, Kucharska, Dutkiewicz, and Rączka (2014) and Rączka, Dudek-Dyduch, Kucharska, and Dutkiewicz (2015). A proposition of the new, expanded structure of ALMM Solver, including a new module named Problem Model Library (PML) has been given by the authors in Dudek-Dyduch and Korzonek (2016). This paper develops the idea of component IT models of problems and, as a result, the component Library of Problem Models. It should be underlined that the paper is a significant extension of the paper ‘ALMM Solver for combinatorial and discrete optimization problems – Idea of Problem Model Library’ (Dudek-Dyduch & Korzonek, 2016) presented at ACIIDS 2016 conference.

The first crucial issue when designing any solver is how to maintain the knowledge about problems to be solved. In other words, how in a formal way should one enter
into the system all the constraints and the quality criterion which define the problem. ALMM Solver operates on formal (mathematical) models of problems that belong to the class of models based on ALMM. Using ALMM, it is possible to develop so-called algebraic-logical (AL) models for a large class of discrete optimization problems as well as combinatorial ones.

The second basic issue is how to implement the AL problem models i.e. what software representation of AL models is to be like? AL models must be implemented in a way facilitating their use in implementation of solving methods and algorithms.

The third question is if there is a simple way to create a software representation of AL models?

This paper contains answers to above questions.

Quite obviously, the literature presents a multitude of discrete optimization problems, general methods of solving them and dedicated algorithms (Błażewicz, Ecker, Pesch, Schmidt, & Węglarz, 2007; Pinedo, 2012). It is so vast an area that it is hard for a typical engineer to navigate, find a book or a paper tackling a specific question and get the needed algorithm. The situation may be improved by the development of software tools that let one solve certain kinds of problems by various known methods. The most popular such tools are based on the Constraint Logic Programming (CLP) approach, focusing on finding feasible solutions (Ligęza, 2012; Rossi, Van Beek, & Walsh, 2006). The problems are modelled there by a finite, predetermined number of variables, their domains and relations between them. However, wide range of actual optimization problems are not suitable for modelling through CLP. Other optimization software tools are based on multi-agent approach (Jędrzejowicz & Ratajczak-Ropel, 2015; Smith & Taylor, 1998), as for example A-Team architecture (Barbucha, Czarnowski, Jędrzejowicz, Ratajczak-Ropel, & Wierzbowska, 2009, 2010; Muller, 1998).

The main difference between ALMM Solver and the two mentioned classes of software tools consists in different methods of problem modelling and consequently different forms of solutions. The solutions for both CLP and the agent approaches are defined by means of a set of variables which should have proper values assigned to. For ALMM-based approach, the solution is defined as a sequence (or a set) of the ‘complex decisions’ wherein a complex decision is presented as an n-tuple of individual decisions that must be or may be taken simultaneously (see Section 2). This way of a solution defining is of a key importance and causes that ALMM is very convenient for modelling dynamic problems.

The deeper analysis of related approaches will be given in another paper to follow.

The objective of the paper is dual, including:

- presenting a method of AL model implementation and component-oriented approach to their development,
- presenting the idea of component Library of Problem Models.

Organization of the paper is as follows. ALMM paradigm of modelling discrete optimization problems is recalled with two sample models given in Section 2. Section 3 presents ALMM Solver extended structure and the architecture of its main computation module. Section 4 presents the Library of Problem Models, focusing on how to design and implement library components. Component technology application to PML designing is presented based on a sample class of scheduling problems with one machine. Software
representation of AL models (proposed by authors in Dudek-Dyduch & Korzonek, 2016) is recalled. Component modelling of alternative and conjunction type of limitations is proposed and discussed. General library source code organization is proposed. Section 5 contains the conclusions.

2. Algebraic-Logical Meta-Model of Multistage Decision Processes

The ALMM is a general model development paradigm for deterministic problems, for which solutions that can be presented as a sequence (or a set) of complex decisions.

The idea of ALMM paradigm was proposed and developed by Dudek-Dyduch (1990, 1992, 2000, 2002) and recalled in many papers among others (Dudek-Dyduch, 2015a). It is also put to use in multiple cases. Based on ALMM, formal models (so-called AL models) may be established for a very broad class of discrete optimization problems from a variety of application areas, thus yielding to the meta-model designation.

In the articles (Dudek-Dyduch, 1990, 1993a, 1993b, 2000, 2002, 2015b, 2015c; Dudek-Dyduch & Dyduch, 1993; Dudek-Dyduch & Kucharska, 2011a) that lay the groundwork for the basic ALMM theory, Dudek-Dyduch E. has presented AL models for discrete manufacturing process control, for logistics problems, knapsack problem and a model for a complex real-life problem encompassing both manufacturing and logistics. Basing on ALMM paradigm, new methods for solving discrete optimization problems have been created too (Dudek-Dyduch, 2000; Dudek-Dyduch & Dutkiewicz, 2013; Dudek-Dyduch & Kucharska, 2011a, 2011b; Dutkiewicz & Dudek-Dyduch, 2014; Kucharska & Dudek-Dyduch, 2014). An overview of the new methods is given in Dudek-Dyduch (2015a).

ALMM provides a structured way of recording knowledge of the goal and all relevant constraints that exist within the problems modelled. In line with this idea, all information is split into pre-defined basic components (elements) with appropriate links, defined both through algebraic and logical formulae.

Using this paradigm, Dudek-Dyduch E. has provided, i.e. in Błażewicz et al. (2007), Dudek-Dyduch (1990) and recall in Dudek-Dyduch (2015a), the definition of two base types of multistage decision processes: a common process, denoted as cMDP and a dynamic process denoted as MDDP. Let us review them briefly.

**Definition 2.1:** Common multistage decision process is a process that is specified by the sextuple $cMDP = (U, X, x_0, f, X_N, X_G)$ where $U$ is a set of decisions, $X$ is a set of states, $f: U \times X \rightarrow X$ is a partial function called a transition function, (it does not have to be determined for all elements of the set $U \times X$), $x_0, X_N \subseteq X, X_G \subseteq X$ are respectively: an initial state, a set of not admissible states and a set of goal states, i.e. the states in which we want the process to take place at the end. Subsets $X_G$ and $X_N$ are disjoint i.e. $X_G \cap X_N = \varnothing$.

Because not all decisions make sense in certain situations, the transition function $f$ is defined as a partial one. As a result, all limitations concerning the decisions in a given state $x$ can be defined in a convenient way by means of so-called sets of possible decisions $U_p(x)$, and defined as: $U_p(x) = \{u \in U: (u, x) \in \text{Dom } f\}$.

The definition quoted below refers to dynamic decision processes, i.e. processes wherein both the constraints and the transition function (and in particular the possible
Decision sets depend on time. Therefore, the concept of the so-called ‘generalized state’ has been introduced, defined as a pair containing both the state and the time instant.

**Definition 2.2:** Multistage dynamic decision process is a process that is specified by the sextuple \( \text{MDDP} = (U, S, S_0, f, S_N, S_G) \) where \( U \) is a set of decisions, \( S = X \times T \) is a set named a set of generalized states, \( X \) is a set of proper states, \( T \subseteq \mathbb{R}_+ \cup \{0\} \) is a subset of non-negative real numbers representing the time instants, \( f: U \times S \rightarrow S \) is a partial function called a transition function, \( f \) does not have to be determined for all elements of the set \( U \times S \), \( S_0 = (x_0, t_0) \), \( S_N \subset S \), \( S_G \subset S \) are respectively: an initial generalized state, a set of not admissible generalized states and a set of goal generalized states, i.e. the states in which we want the process to take place at the end. Subsets \( S_G \) and \( S_N \) are disjoint i.e. \( S_G \cap S_N = \emptyset \).

The transition function is defined by means of two functions, \( f = (f_s, f_t) \) where \( f_s: U \times X \times T \rightarrow X \) determines the next state, \( f_t: U \times X \times T \rightarrow T \) determines the next time instant. It is assumed that the difference \( \Delta t = f_t(u, x, t) - t \) has a value that is both finite and positive.

Just like in Definition 2.1, the transition function \( f \) is defined as a partial one. As a result, all limitations concerning the decisions in a given state \( s \) can be defined in a convenient way by means of so-called sets of possible decisions \( U_p(s) \), and defined as: \( U_p(s) = \{ u \in U : (u, s) \in \text{Dom} f \} \).

For both defined types of the multistage decision processes, in the most general case, sets \( U \) and \( X \) may be represented as a Cartesian product \( U = U^1 \times U^2 \times \cdots \times U^m \), \( X = X^1 \times X^2 \times \cdots \times X^n \) i.e. \( u = (u^1, u^2, \ldots, u^m) \), \( x = (x^1, x^2, \ldots, x^n) \). In particular, \( u^i, i = 1, 2, \ldots, m \) represent separate decisions that must be or may be taken at the same time (for Definition 2.2) or at the same stage (for Definition 2.1) and relate to particular objects. There are no limitations imposed on types of the sets; in particular, they do not have to be numerical. Thus, values of particular co-ordinates of a state or a decision may be names of elements (symbols) as well as some objects (e.g. finite set, sequence etc.).

The sets \( X_N, S_N, X_G, S_G \) and \( U_p \) are formally defined with the use of both algebraic and logical formulae, hence the algebraic-logic model descriptor.

Based on the meta-model recalled herein, AL models may be created for individual problems consisting of seeking admissible or optimal solutions. In case of an admissible solution (admissibility problem), an AL model is equivalent to a suitable multistage decision process, hence it is denoted as process \( P \). An optimization problem then is denoted as a \((P, Q)\) pair where \( Q \) is a criterion. An optimization task (instance of the problem) is denoted as a \((P, Q)\). A sequence of consecutive states from the initial state to a final state (goal or not admissible or blind), computed by transition function forms a process trajectory.

The structured character of AL model is very useful. It is this property that allows for modifying the AL model easily, when some of the limitations change. The simple examples of scheduling problem AL models below illustrate this.

### 2.1. Examples of AL models

As it is well known, scheduling problems are denoted by \( \alpha|\beta|\gamma \) notation (Błażewicz et al., 2007), where \( \alpha \) represent information on machines, \( \beta \) – restrictions referring to the jobs and \( \gamma \) – a criterion.
Example 2.1: Single machine problem 1||\(C_w\).

A finite set of jobs is given, represented by natural numbers \(Z = \{1, 2, \ldots, n\}\), \(n\) – number of jobs. There is also given a function \(p:Z \rightarrow \mathbb{R}^+\) that determines the processing times and a function \(w:Z \rightarrow \mathbb{R}^+\) assigning weights to individual jobs. We have to find an order of the job performance so weighted sum of jobs completion times \(C_w = \sum_{j=0}^{n} w_j C_j\) is at minimum, where \(w_j, C_j\) are weight and completion time of the \(j\)th job respectively.

Let us present the elements of AL model of the problem.

The proper state \(x\) represents the set of completed jobs. Thus, the set of a proper state \(X\) is the set of all job subsets \(X = 2^Z\).

The decision consists in determining the next job to be performed by the machine. Thus, the set of decisions \(U\) is equal to a set of tasks \(U = Z\).

There are no not admissible states, so the set of not admissible states \(S_N = \emptyset\).

The set of goal states \(S_G = \{(x, t): x = Z\}\) – it denotes that the process is in a goal state if all the jobs are completed.

The initial state \(s_0 = (x_0, t_0) = (\emptyset, 0)\).

The transition function \(f:U \times X \times T \rightarrow X \times T\) is determined as follows:

\[
U_p(x, t) = Z \setminus x, \quad \text{thus} \quad U_p(s_0) = Z
\]

\[
f_p(u, x, t) = x \cup \{u\}, \quad f_t(u, x, t) = t + p(u)
\]

The set of possible decisions in the \(x_{i+1}\) state may be calculated by a simple modification of the previous set: \(U_p(s_{i+1}) = U_p(s_i) \setminus \{u_i\}\).

From the \(U_p(x, t)\) set definition and the fact that \(S_N = \emptyset\) it follows that any sequence of decisions \(\bar{u}\) corresponding to a permutation of jobs is an admissible solution.

Example 2.2: Single machine problem 1|\(C_j \leq d_j|C_{\text{max}}\)

We shall modify the job presented in Example 2.1 setting deadlines of the jobs. Let the function \(d:Z \rightarrow \mathbb{R}^+\) determine the deadlines of individual jobs.

The set of states \(X\), the set of decisions \(U\), the set of generalized goal states \(S_G\) and the initial state \(s_0\) are defined in the same way as previously, thus: \(X = 2^Z\), \(U = Z\), \(S_G = \{(x, t): x = Z\}\).

In turn, the definition of the set \(S_N\) changes. It is now defined by the set of all pairs \((x, t)\) such that the time is larger or equal to the deadline of any job, while the job does not belong to the set of the finished jobs.

\[
S_N = \{(x, t): \exists i \in Z, i \notin x, d(i) \leq t\}
\]

The transition function \(f\) is defined as in Example 2.1. Whether a state is admissible is indeed dependent on time. That is why not every permutation of jobs shall correspond to an admissible sequence of decisions.

3. Extended structure of ALMM Solver

The task of ALMM Solver is to provide solutions (exact or approximate) for combinatorial and discrete optimization problems or indicate that a solution has not been found. As it was stated, a solution has form of a sequence (or set) of decisions.

The initiative structure of ALMM Solver has been given in Dudek-Dyduch et al. (2014) and Raźcza et al. (2015). The structure has been extended in Dudek-Dyduch & Korzonek
(2016) where a new module, namely PML has been introduced. Presented in Figure 1 is the modified, extended structure, including PML. (Actually, the modules presented in Figure 1 only constitute a part of the real ALMM Solver structure, but they are sufficient for the purposes of this paper.)

Knowledge of problem and its instance is forwarded from PML or Algebraic-Logic Models (ALM) Modeller to the main Solver module carrying out the computations.

On this stage of ALMM Solver development, SimOpt operates as its computation module. Its computations are controlled by algorithms stored in the Method/Algorithm Module. Presently, only the so-called constructive algorithms i.e. based on a state tree generation are implemented. Computation Module (SimOpt) task is to intelligently construct the state transition tree (either in its entirety, or only a part of it) by generating one or more trajectories (parts of trajectories). The solver database stores all generated states of the trajectories maintaining the structure of the tree. The initial structure of the Computation Module (Dudek-Dyduch et al., 2014) slightly modified is presented in Figure 2 while its detailed description is given in Dudek-Dyduch et al. (2014) and Rączka et al. (2015). Due to the limited length of the paper, it must be omitted here.

The objective of ALM Modeller module is to initiate and forward information about the AL model of a problem and its instance to the Computation Module. The information can also come directly from the PML. The model needs to take a form understandable to the Computation Module i.e. it should have a software form (IT model) (presented in Section 4). Both ALM Modeller and PML are linked to the User Interface and Computation Module. The models can be created in a few ways, including:

- downloading a ready model from PML
- building a model from PML components
- developing a new model from scratch and implementing it in a proper programming language.

Results Interpreter (Dudek-Dyduch et al., 2014) gives the users the solution (one or many) or return an information that a solution was not found. In addition, another
important feature of this module is that it contains the visualization of results. In the future, the module will also provide a mean to share the results of the analysis through external API. It will allow one to integrate ALMM Solver with other systems, for example, with manufacturing planning systems (Dudek-Dyduch, 2002), and to use Business Intelligence tools. Additionally, the module will provide the possibility to define various ways of analysing results obtained from a series of computations for the same or different instances of a problem.

4. Library of Problem Models

It is assumed that the library must be characterized by flexibility and ability to adjust to certain problem subsets. It is necessary to ensure the opportunity of its gradual expansion too, and the development of simple rules for adding new elements. The Library provides software representation of AL models to various Solver modules. At this point, any application or module supported by the Library will be referred to as a client application.

C# was selected for library implementation because of its object-oriented character, capacities, ease of use and availability of multiple class libraries for various purposes, as well as its reliability.
4.1. Software representation of AL models

The most important issue is to adopt an appropriate way for implementing the problem model that is to design a software representation of AL models. As mentioned earlier, process $\mathbf{P}$ is the formal (mathematical) model of admissibility problem, while a $(\mathbf{P}, \mathbf{Q})$ correspond to optimization problem. In this section, we only consider admissibility problems, and thus discuss software models of processes $\mathbf{P}$ only.

The AL model of a problem is implemented as a class named $c$Problem. Problem instances will be matched by objects from that class.

The AL model indicates elements of the problem that need to be defined in the $c$Problem class, i.e. definitions of: decision and state structures, initial state, transition function, sets of possible decisions for given states, set of not admissible states, set of goal states and definition of problem data structures. The interface of the class that models a problem should provide methods related to the said elements:

- $c$State GetInitialState() – determining the initial state
- $c$State GetNextState($c$State, $c$Decision) – transition function
- bool IsGoalState($c$State) – verifying if a given state is a target state
- bool IsNotAdmissibleState($c$State) – verifying if a given state is an not admissible state
- bool IsPossibleDecision($c$State, $c$Decision) – verifying if a given decision belongs to the set of possible decisions.

The C# code below is a declaration of interfaces matching individual elements of problem definitions and grouping them in a whole interface for a $c$Problem class.

```csharp
public interface iPossibleDecisionVerifier
{
    bool IsPossibleDecision ($c$State CurrentState, $c$Decision CurrentDecision);
}
public interface iStateInitializer
{
    $c$State GetInitialState();
}
public interface iStateTransformer
{
    $c$State GetNextState ($c$State CurrentState, $c$Decision CurrentDecision);
}
public interface iNotAdmissibleStateVerifier
{
    bool IsNotAdmissibleState ($c$State CurrentState);
}
public interface iGoalStateVerifier
{
    bool IsGoalState ($c$State CurrentState);
}
public interface iProblem: iPossibleDecisionVerifier, iStateInitializer, iStateTransformer, iNotAdmissibleStateVerifier, iGoalStateVerifier,
{
}
```

4.2. Component-based structure of library

A developer has much freedom in designing and creating classes, using class inheritance and composition. In order to maintain a clear structure enabling easy Library expansion, it
is important to categorize class sets constituting Library components. Dividing them into concrete classes, abstract classes, association classes and ones not matching any of these categories seems insufficient. The UML standard versions 2.0 and above (OMG) allow the definition of own types of the so-called generalization sets describing classification criteria for base and inherited classes.

Taking into account the class availability for library clients and purposes of the class use, the following categorization of classes into distinct groups is proposed:

- utility (boundary) classes – the only classes available to the client application, to be used to build modelling objects, e.g. problem, state and decision. As such they cannot be abstract classes;
- atomic, elementary classes – responsible for exactly one element of the AL model, these implement in a strictly defined way one or more methods related to that element. They constitute the basic components building other library classes;
- composite classes – classes with use similar to elementary classes, but combining atomic classes. They contain several components of respective elementary classes. This way it is possible to use ready implementation of atomic classes in order to create a new method made available by a given composite class (these classes are used because of no multiple inheritance available in C#) and
- design classes – during implementation of solutions for complex problems, these can be helpful in developing other classes. They can constitute base classes from which other classes may inherit. They can also be components of composite classes or design classes.

Once defined, the utility, atomic, composite and design classes cannot be modified by library users. In order to expand the library and provide support for new types of problems, the user should be able to create new classes and use already existing ones. Only utility classes are available to client applications. Only design classes can be abstract classes, but it is not necessary to define them in this way. Elementary and composite class objects can be components of other composite, design or utility classes (Figure 3).

It is assumed that the library of problems will be extended. Every problem being modelled corresponds to appropriate set of software classes. New classes must be added to the library when modelling new problems. It is desirable that the existing code is not duplicated. It is therefore necessary to check whether one can use ready-made classes to implement a new target. In view of the growing number of library classes, it will be increasingly difficult to control the code. One possible way to handle this difficulty is to group classes according to certain categories. Software classes can be associated with a certain distinguished sets of problems and it is the most general category of the division. A much more detailed category can be established by associating software classes with certain elements of the AL model, i.e. definition of goal state, not admissible state and so on. Dividing the library classes for atomic (elementary), boundary (utility), composite and design classes is an additional criterion and thus simplify the management of library items.

4.3. Utility and atomic classes

The analysis of AL models for a given area such as task scheduling, graph problems and others shows that definitions of some AL model elements are identical (Dudek-Dyduch,
As a result, in classes modelling different problems, we may identify the same attributes and methods. That fact implies the idea of establishing utility class `cProblem` from relevant atomic classes because atomic classes correspond to some parts of a reusable code.

The design pattern ‘Strategy’ (Gamma, Helm, Johnson, & Vlissides, 1995) may be used to construct the class `cProblem`.

This way of building utility classes will be presented on the example of five consecutive job-scheduling problems with a single machine. The problems are: \( \prod_1 = 1 \parallel C_{\text{max}} \), \( \prod_2 = 1 \mid C_j \leq d_j \mid C_{\text{max}} \), \( \prod_3 = 1 \parallel \text{prec} \mid C_{\text{max}} \), \( \prod_4 = 1 \mid r_j \mid C_{\text{max}} \) and \( \prod_5 = 1 \mid C_{\text{max}} \leq \tilde{C} \mid \sum C_j \), where \( C_j \) – the completion time of the \( j \)-th job, \( C_{\text{max}} \) – the criterion of total completion time, \( \tilde{C} \) – the parameter determining maximal total completion time, \( \sum C_j \) – the criterion of sum of job completion times, \( d_j, r_j \) denote restrictions referring to the job due dates and release time respectively and \( \text{prec} \) denotes existence of preceding constraints in the set of jobs. AL models of these problems have been developed by Dudek-Dyduch E. and presented in Dudek-Dyduch (1990, 1993a, 1993b, 2002). They are characterized by identical definitions of state and decision structures as well as the definition of initial state \( s_0 \). Definitions of not admissible state set \( S_{nv} \), final state set \( S_{Gr} \), transition function \( f \) and sets of possible decisions \( U_p(s) \) for the problem \( \prod_1 = 1 \parallel C_{\text{max}} \) shall be assumed as default. Some of the definitions above can be used without modifications for the remaining four problems (Table 1).

In the table above, fields marked with the ‘default(\( \prod_i \))’ tag mean definitions identical to formulas for the problem \( \prod_i \). Fields marked with the ‘specific(\( \prod_i \))’ tag mean a definition specific to a given problem \( \prod_i \). For the problem \( \prod_2 \) it is necessary to develop a specific
definition of $S_N$ set. For the problem $\prod_3$ a new definition of sets $U_p(s)$ is necessary. In case of the problem $\prod_4$ the transition function has to be redefined. Only for the problem $\prod_5$ two definitions need to be changed: the definition of sets $S_N$ and $S_G$. The latter one will take a different form from the analogical definition for the problem $\prod_2$. This means three definitions universal for all the problems need to be developed together with four default ones and five specific to individual problems.

In the example considered, each problem element definition is matched by a single atomic class implementing one or more methods. Thus, 12 elementary classes will emerge. Using these classes, it is possible to develop the software models of all 5 given problems with reusable code.

With such variety of possible constraints, there is a need to build classes modelling problems appropriate for each case. Regardless of using design classes or deciding not to use them, the developer will then have to create a big number of utility classes modelling the problems. There is a way, though, to develop a single universal class for the problems in question. It must be noted that, in each case, the structures, interfaces and some class methods will be identical. Only some methods will differ depending on the case. In that case, a ‘strategy’ design pattern (Martin & Martin, 2006) may be used, creating a single universal class modelling the problem. The behaviour of its objects will vary, but it can be controlled, thanks to replaceable components, while the new class will retain a constant structure. It is necessary to develop appropriate rules for the selection of elementary or composite classes based on specific problem instances. A single utility class developed this way will be an appropriate IT model for multiple problems.

4.4. Modelling conjunction and alternative type constraints

The library should enable building problem classes with several constraints occurring simultaneously. Constraints specified in the $\alpha|\beta|\gamma$ notation may concern different AL model elements or the same element. Moreover, the constraints are ‘translated’ to logical formulas defining individual AL model elements.

For example, for the new problem $\prod_6 = 1|r_j, C_j \leq d_j|C_{\text{max}}$ the constraints pertain to independent problem definition elements. The constraint concerning release times $r_j, j = 1, 2, \ldots, n$ modifies the definition of $U_{p}(s)$ sets, whereas the constraint $C_j \leq d_j, j = 1, 2, \ldots, n$ defines the $S_N$ set. Thus, previously defined elementary classes may be used to create a utility class for the problem $\prod_6$.

In case of the problem $\prod_7 = 1|C_j \leq d_j, C_{\text{max}} \leq \bar{C}_1|\sum C_j$ both constraints refer to the same AL model element, namely $S_N$. A state is deemed not admissible if at least one of these constraints is violated. For the problem $\prod_7$ a new composite class needs to be developed, containing elementary class objects specific for both problems: $\prod_2$ and $\prod_5$.

---

**Table 1.** Default or specific definitions of problem elements.

|   | $S_N$  | $S_G$  | $t$   | $U_p(s)$ |
|---|--------|--------|-------|---------|
| $\prod_1$ | default($\prod_1$) | default($\prod_1$) | default($\prod_1$) | default($\prod_1$) |
| $\prod_2$ | specific($\prod_2$) | default($\prod_1$) | default($\prod_1$) | default($\prod_1$) |
| $\prod_3$ | default($\prod_1$) | default($\prod_1$) | default($\prod_1$) | specific($\prod_3$) |
| $\prod_4$ | specific($\prod_3$) | default($\prod_1$) | specific($\prod_3$) | default($\prod_1$) |
| $\prod_5$ | specific($\prod_3$) | specific($\prod_3$) | default($\prod_1$) | default($\prod_1$) |
Appropriate methods from both objects comprising the newly developed composite class may be used in implementation of the new method to verify if a given state is not admissible. However, the manner of developing a composite class of already defined elementary classes needs to be flexible. Let us discuss this question.

In general, the sets \( S_N, S_G \) and \( U_p \) may be defined by means of an alternative or conjunction of logical formulas \( \phi_i = 1, 2, \ldots, k \), for example \( S_N = \{ s : \varphi_1(s) \lor \varphi_2(s) \lor \cdots \lor \varphi_k(s) \} \) (see the problem \( \prod \)).

Let us assume that elementary classes \( A_i, i = 1, 2, \ldots, k \) are responsible for verification if the \( \phi_i = 1, 2, \ldots, k \) formulas are true. Thus, each of the elementary classes \( A_i \) implements the \texttt{IsNotAdmissibleState} method from the \texttt{iNotAdmissibleStateVerifier} interface corresponding to the \( \phi_i \) formula.

The developer aims to create a new class implementing the same interface and not admissible state verifier accordingly to definition \( S_N \). The desired effect can be achieved by means of several manners. The simplest one consists in creating composite class including component objects from classes \( A_i, i = 1, 2, \ldots, k \) and implementing the \texttt{IsNotAdmissibleState} method as a logical alternative of the results returned by object methods from classes \( A_i \). The above method of implementing composite class \( C_1 \) is simple, but not particularly flexible.

It should be remembered that the Library of Problem Models is to enable developing problem models with various constraints. Then, it may prove to be necessary to create numerous different composite classes with a similar structure and realizing the same method in an analogical manner. The solution in Figure 4 comprising direct combining of concrete atomic classes with composite classes causes the latter to bear the responsibility both for the selection of atomic classes and realizing the alternative. As a result, such a solution causes the danger of copying a fragment of the source code responsible for realizing the constraints alternative in all composite classes modelling the state verifier. Then an attempt at broadening the methods realizing alternatives, e.g. by correctness check of elements (or factors for conjunction) and exceptions handling, will have the effect of the necessity to introduce the same changes in all analysed composite classes.

The authors propose a solution providing the possibility to conveniently create necessary classes realizing the alternative of constraints and avoiding the copying of existing code. The proper result may be achieved through using DRY (do not repeat yourself) rule (Hunt & Thomas, 1999) and Single Responsibility rule (DeMarco, 1979; Martin &

---

**Figure 4.** UML diagram of sample composite class use for two formulas.
Martin, 2006; Page-Jones, 1988) – one of SOLID principles (Martin, 2008; The Principles of OOD; Uncle Bob Consulting LLC).

For the sake of simplicity of further discussion, let us limit the number of formulas in the definition of $S_N$ set to two, i.e. $S_N = \{s: \phi_1(s) \lor \phi_2(s)\}$. In the proposed solution, the set of classes $\{A_1, A_2, C_1, U_1\}$ (Figure 4) is expanded by auxiliary design classes $D_1$ and $D_2$. Relations between individual classes have been presented in Figure 5, while their role shall be discussed below.

A certain composite class $C_2$ is sought, which may be incorporated in a utility class $U_1$ modelling the problem and, at the same time, will allow to realize an alternative of any constraints impacting the form of the set of not admissible states $S_N$. A class being a generalization of classes modelling the analysed constraints is an IT model of any constraint. Thus, an abstract design class $D_1$ has been defined as the base class for the atomic classes $A_1$ and $A_2$. The three above-mentioned classes inherit the same interface $iNotAdmissibleStateVerifier$; however, only the atomic classes $A_1$ and $A_2$ implement the $IsNotAdmissibleState$ method, whereas the design class $D_1$ solely declares that some of its child classes shall realize the above-mentioned method.

Next, a design class $D_2$ was defined, which also inherits the $iNotAdmissibleStateVerifier$ interface. The given class is responsible solely for realizing the alternative of two constraints associated with the definition of the not admissible states $S_N$. The developed model of constraints alternative should be independent of atomic classes modelling specific definitions and should be abstract by assumption. For this purpose, the $D_2$ class has been provided with two elements, that is references to the $D_1$ class elements. When the application is running, these references will be associated with actual objects of atomic classes, e.g. $A_1$ and $A_2$. It should be remembered that the $D_2$ class definition does not specify which atomic classes represent its individual elements. The design class $D_2$ is responsible solely for the manner of realizing the $IsNotAdmissibleState$ method. This takes place through using analogical methods of its elements and, in the analysed example, through returning the alternative of results of the above-mentioned methods.

The composite class $C_2$ inherits the elements and implementation of the $IsNotAdmissibleState$ method after the $D_2$ class and is responsible solely for indicating specific atomic classes modelling the constraints, which alternative we want to obtain. The composite class $C_2$ may be used for developing the utility class $U_1$.

Conjunction and other combinations of constraints may be modelled in a manner similar to that described above.

Figure 5. UML diagram of sample composite class and design class use.
4.5. General library code organization

An IT project source code is frequently divided into smaller parts recorded in files, which in turn are grouped into folders according to an accepted criterion. A similar division has been adopted for the problem library code.

Due to the expected expansion of the library classes number, the engineers dealing with library maintenance and development may face increasing difficulties in searching for the defined classes. This operation will be necessary to be able to define what problems are supported by the library or which classes may be reused for defining a problem of a new type and which part of the problem model should be programmed from scratch. The division of all library classes into smaller subsets will make this job easier to some extent. To this purpose, the model library code will be divided into files and folders in such a way that it corresponds to the AL model structure. Folders grouping code fragments defining individual elements of AL model and general problem structure will be placed on the top of the folder hierarchy. Let us call the above-mentioned folders master folders. Names of the master folders will unequivocally suggest the problem definition element to which the files placed in them pertain. Thus, all classes associated with the definition of e.g. the goal state set $S_G$ will be grouped in the folder named Goal State. Lower level in the folder hierarchy will be constituted by subfolders, grouping files with classes’ definitions: atomic, composite, design and border/utility. The adopted structure of folders, files and classes may be illustrated in the composition (Figure 6).

Figure 6 presents the source code organizational chart of the library, where the division of classes per the association with a definition element defines ‘vertical’ organization, whereas the division of classes per the manner of utilization defines ‘horizontal’ organization.

4.6. Developing the library

In case of a need to expand the library by a possibility to handle a new problem, it should be verified if the available library classes are sufficient to model each of the elements of AL model of the given new problem.

|    | $P$ | $f$ | $U$ | $U_P$ | $S$ | $S_G$ | $S_N$ | $S_0$ |
|----|-----|-----|-----|-------|-----|-------|-------|-------|
| Atomic       | [Diagram of Atomic Classes] | [Diagram of f Classes] | [Diagram of U Classes] | [Diagram of $U_P$ Classes] | [Diagram of S Classes] | [Diagram of $S_G$ Classes] | [Diagram of $S_N$ Classes] | [Diagram of $S_0$ Classes] |
| Boundary /Utility | [Diagram of Boundary Classes] | [Diagram of Utility Classes] | [Diagram of Composite Classes] | [Diagram of Design Classes] |

Figure 6. Library classes division.
Thanks to the adopted library code structure, the developer may perform a methodical overview of classes for individual problem definition elements. A few rules facilitating the overview performance may be indicated. Border/utility classes are invariable and no changes shall pertain to the classes of a given type. It should be noticed that there is no justification for mixing atomic or composite classes of different subsets associated with different problem definition elements. That is why one should focus rather on searching classes belonging to a given master folder. Only in case of design classes, searching folders of this type of classes in other master folders may be justified.

A typical case of library development may take the following course: for each problem definition element, it should be verified if a given element may be modelled by means of existing atomic classes. If yes, then whether it is possible to use existing composite classes. Any missing problem model elements must be modelled as new atomic classes and, if needed, a new composite class must also be created. Moreover, the use of design classes should be considered. If utilizing them proves to be useful, then it should be checked if existing design classes may be used. It is not impossible that useful design classes may also be found in other master folders of classes.

5. Conclusion

The paper presents the new results of research concerning ALMM Solver, the tool for solving various discrete optimization problems, especially non-deterministic polynomial-time hard problems. The solver uses an ALMM and its theory.

ALMM enables a unified approach to developing formal models of discrete optimization problems (AL models) and recording problem-related knowledge. The AL models and their software representations are stored in PML. The structured character of ALMM implies the structured character of AL models. It is this property that allows for component technology applications to design PML. The paper presents the library, focusing on how to design and implement its components. Authors propose general library code organization and discussed component implementation of problem constraints of different kinds. The consideration is illustrated by means of the sample class of scheduling problems.

To conclude, introducing the modified structure of ALMM Solver that contains the Library of Problem Models, proposing component technology for implementation of the library and proposing a structure and rules for developing the library are stated as the paper’s main contribution.

Disclosure statement

No potential conflict of interest was reported by the authors.

Notes on contributors

Sławomir Korzonek is a master of science graduate of AGH, University of Science and Technology in Kraków since 2004, C++ programmer in ATSI S.A. for 13 years. Interests include object oriented languages, systems modeling, scheduling problems and history.

Ewa Dudek-Dyduch is a Professor and part of the academic and teaching staff of AGH, University of Science and Technology in Kraków and author of the ALMM paradigm ant its theory. The area of interest includes optimization problems, simulation systems, artificial intelligence, and manufacturing processes.
References

Barbucha, D., Czarnowski, I., Jędrzejowicz, P., Ratajczak-Ropel, E., & Wierzbowska, I. (2009). e-JABAT – An implementation of the web-based A-team. In N. T. Nguyen & L. C. Jain (Eds.), Intelligence agents in the evolution of Web and applications. Studies in computational intelligence (Vol. 167, pp. 57–86). Heidelberg: Springer.

Barbucha, D., Czarnowski, I., Jędrzejowicz, P., Ratajczak-Ropel, E., & Wierzbowska, I. (2010). Influence of the working strategy on A-team performance. In E. Szczerbicki & N. T. Nguyen (Eds.), Smart information and knowledge management. Studies in computational intelligence (Vol. 260, pp. 83–102). Heidelberg: Springer.

Błażewicz, J., Ecker, K., Pesch, E., Schmidt, G., & Węglarz, J. (2007). Handbook on scheduling. Berlin: Springer.

DeMarco, T. (1979). Structured analysis and system specification. East Lansing: Prentice Hall.

Dudek-Dyduch, E. (1990). Formalization and analysis of problems of discrete manufacturing processes (Scientific bulletin of AGH University. Automatics vol. 54) (in Polish).

Dudek-Dyduch, E. (1992). Control of discrete event processes – branch and bound method. In Proc. of IFAC/IFORS/IMACS symposium large scale systems: Theory and applications (Vol. 2, pp. 573–578). Chinese Association of Automation.

Dudek-Dyduch, E. (1993a). Discrete determinable processes – compact knowledge-based model (Notas de Matematica No 137). Merida: Universidad de Los Andes.

Dudek-Dyduch, E. (1993b). Problems of knowledge representation in expert system aided control of DPP [Inżynieria Wiedzy i Systemy Ekspertowe, Prace II Krajowej Konferencji, tom I] (pp. 147–154). Wrocław: Politechnika Wrocławska (in Polish).

Dudek-Dyduch, E. (2000). Learning based algorithm in scheduling. Journal of Intelligent Manufacturing (JIM), 11(2), 135–143 (accepted to be published 1998).

Dudek-Dyduch, E. (2002). Information systems for production management. Kraków: Wyd. Poldex (in Polish).

Dudek-Dyduch, E. (2015a). Algebraic logical meta-model of decision processes – new metaheuristics. In L. Rutkowski, M. Korytkowski, R. Scherer, R. Tadeusiewicz, L. A. Zadeh, & J. M. Żurada (Eds.), ICAISC 2015, part I. LNCS (Vol. 9119, pp. 541–554). Cham: Springer.

Dudek-Dyduch, E. (2015b). Modeling manufacturing processes with disturbances – a new method based on algebraic-logical meta-model. In L. Rutkowski, M. Korytkowski, R. Scherer, R. Tadeusiewicz, L. A. Zadeh, & J. M. Żurada (Eds.), ICAISC 2015, part II. LNCS (Vol. 9120, pp. 353–363). Cham: Springer.

Dudek-Dyduch, E. (2015c). Modeling manufacturing processes with disturbances – two-stage AL model transformation method. 20th International conference on methods and models in automation and robotics, MMAR proceedings, Miedzyzdroje, Poland, pp. 782–787.

Dudek-Dyduch, E., & Dutkiewicz, L. (2013). Substitution tasks method for discrete optimization. In L. Rutkowski, M. Korytkowski, R. Scherer, R. Tadeusiewicz, L. A. Zadeh, & J. M. Żurada (Eds.), ICAISC 2013, part II. LNCS (Vol. 7895, pp. 419–430). Berlin: Springer.

Dudek-Dyduch, E., & Dyduch, T. (1993). Formal approach to optimization of discrete manufacturing processes. In M. H. Hamza (Ed.), Proc. of the twelfth IASTED Int. conference modelling, identification and control (pp. 419–430). Zurich: Acta Press.

Dudek-Dyduch, E., & Korzonek, S. (2016). ALMM solver for combinatorial and discrete optimization problems – idea of problem model library. In N. T. Nguyen, B. Trawiński, H. Fujita, & T. -P. Hong (Eds.), ACIDS 2016 part I. LNAI (Vol. 9621, pp. 459–469). Heidelberg: Springer.

Dudek-Dyduch, E., & Kucharska, E. (2011a). Learning method for co-operation. In P. Jędrzejowicz, N. T. Nguyen, & K. Hoang (Eds.), ICCCI 2011 part II. LNCS (Vol. 6923, pp. 290–300). Heidelberg: Springer.

Dudek-Dyduch, E., & Kucharska, E. (2011b). Optimization learning method for discrete process control. In IICNCO 2011 (vol. 1, pp. 24–33). Setúbal: SciTePress.

Dudek-Dyduch, E., Kucharska, E., Dutkiewicz, L., & Rączka, K. (2014). ALMM solver – a tool for optimization problems. In Artificial intelligence and soft computing (pp. 328–338). Cham: Springer.

Dutkiewicz, L., & Dudek-Dyduch, E. (2014). Substitution tasks method for co-operation. In Recent developments in computational collective intelligence (pp. 103–113). Cham: Springer.
Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design patterns: Elements of reusable object-oriented software*. Reading: Addison Wesley.

Hunt, A., & Thomas, D. (1999). *The pragmatic programmer: From journeyman to master*. Reading: Addison-Wesley Professional.

Jędrzejowicz, P., & Ratajczak-Ropel, E. (2015). Reinforcement learning strategy for solving the MRCPSP by a team of agents. In R. Neves-Silva, L. C. Jain, & R. J. Howlett (Eds.), *Intelligent decision technologies, smart innovation, systems and technologies* (Vol. 39, pp. 537–548). Cham: Springer.

Kucharska, E., & Dudek-Dyduch, E. (2014). Extended learning method for designation of cooperation. In *Transactions on computational collective intelligence XIV* (pp. 136–157). Berlin: Springer.

Łęga, A. (2012). Improving Efficiency in constraint logic programming through constraint modeling with rules and hypergraphs. In *Federated Conf. on computer science and information systems* (pp. 101–107). Wrocław: IEEE Computer Society Press.

Martin, C. R. (2008). *Clean code*. Upper Saddle River: Prentice Hall.

Martin, C. R., & Martin, M. (2006). *Agile principles, patterns, and practices in C#*. Upper Saddle River: Prentice Hall.

Müller, J. (1998). The right agent (architecture) to do the right thing. In *Proceedings of the fifth international workshop on agent theories, architecture and languages, intelligent agents V*. OMG: *OMG: Unified modeling language version 2.5*. Retrieved from http://www.omg.org/spec/UML/Current

Page-Jones, M. (1988). *The practical guide to structured systems design* (2nd ed.). Upper Saddle River: Prentice Hall.

Pinedo, M. L. (2012). *Scheduling. Theory, algorithms and systems*. Berlin: Springer.

Rączka, K., Dudek-Dyduch, E., Kucharska, E., & Dutkiewicz, L. (2015). ALMM solver: The idea and the architecture. In L. Rutkowski, M. Korytkowski, R. Scherer, R. Tadeusiewicz, L. A. Zadeh, & J. M. Żurada (Eds.), *ICAISC 2015, part II. LNCS* (Vol. 9120, pp. 504–514). Cham: Springer.

Rossi, F., Van Beek, P., & Walsh, T. (2006). *Handbook of constraint programming*. New York: Elsevier.

Smith, R. E., & Taylor, N. (1998). A framework for evolutionary computation in agent-based systems. In C. Looney & J. Castaing (Eds.), *Proceedings of the 1998 international conference on intelligent systems* (pp. 221–224). Buffalo: ISCA Press.

Uncle Bob Consulting LLC. *Getting a SOLID start*. Retrieved from https://sites.google.com/site/unclebobconsultingllc/getting-a-solid-start