Combination of DSL and DCSP for Decision Support in Dynamic Contexts

Boris Ulitin(✉), Eduard Babkin, and Tatiana Babkina

National Research University Higher School of Economics,
Nizhny Novgorod, Russia
{bulitin, eababkin, tbabkina}@hse.ru

Abstract. The article is related to the problem of decision support in dynamic business contexts where conditions, values and goals frequently change over time, and users should participate continuously in the problem definition. In our research we explore an opportunity to organize and simplify decision support during complex resource allocation processes by combining domain specific languages (DSL) and distributed constraint satisfaction techniques (DCSP). We describe a particular domain-specific language and the corresponding semantic model in terms of a newly proposed DSL&DCSP framework. Applicability of the framework is demonstrated using a real-life example of resource allocation process in the railway transportation.

Keywords: Domain-specific language · Constraints satisfaction · Railway transportation · Decision support

1 Introduction

Currently in many domains continuous changes of the problem context require repeatable identification and reformulation of the models, used during the process of problems solving. Correspondingly, decision support in dynamic contexts becomes one of the most significant research challenges.

The resource allocation problem represents a business-relevant example of decision support in dynamic contexts. There are many well-known methods for resource allocation [1], however most of them require a stable context of the problem being solved. As a result, they are not applicable in the cases when frequent changes are possible in the solving procedure and conditions, as well as participation of stakeholders is required. To overcome such a drawback Hodgson, Fernandez-Lopez and Gomez-Perez offer to separate the model, which is responsible for the solving procedure, and a mechanism, which allows controlling this model [2–4].

Hodgson and Fernandez-Lopez [2, 3] define the trees and different kinds of state machines as the best choice for the specification of the solving model. Other researchers [4, 5, 7] offer special languages as the most effective choice for that task. We support the second direction, because in our opinion domain-specific languages (DSL) represent the most convenient, organic and clear method for controlling the dynamically changed context.

Our research presupposes a new approach to connect a user-defined specification in terms of DSL and a mathematical specification of the problem for automated constraints

V. Řepa and T. Bruckner (Eds.): BIR 2016, LNBIP 261, pp. 159–173, 2016.
DOI: 10.1007/978-3-319-45321-7_12

solvers. Following that approach, we propose a DSL&DCSP framework, which automatically translates DSL specifications of a certain resource allocation problem to a specification of a distributed constraint satisfaction problem (DCSP) [8]. For demonstration of our approach current research contained implementation of a software prototype based on the multi-agent DCSP-solver Choco [9] and evaluation the prototype in a real case of railways allocation.

There was an attempt by Prud'homme [18] to use a similar structure of the framework for translations of DSL specifications to constraints. But in contrast to our research, that example uses DSL for specialization of CSP from a general class of mathematical problems. That DSL does not correspond to the already existing, defined CSP model, moreover it is not oriented toward any specific practical application.

The article describes our results as follows. In Sect. 2 we give some facts from the theory of design of domain-specific languages and its using in the course of the solving constraint satisfaction problems. Section 3 describes a high level design and technologies of the proposed DSL&DCSP framework. Section 4 demonstrates in the case of the railway allocation problem how our approach is applied in the real-life situations. In Sect. 5 we discuss the results obtained. We conclude the article with analysis of the results and specification of the future researches.

2 Background

2.1 Definition and the Classification of the DSL

A domain-specific language (DSL) is a computer language specialized to a particular application domain. This is in contrast to a general-purpose language, which is broadly applicable across domains, and lacks specialized features for a particular domain [5].

Any DSL satisfies a number of conditions. The first one addresses the structure of the language. In this context it means that DSL includes not only some amount of commands, but their dependences also. The same command, paired with another language element, can carry a different meaning. The second condition specifies that DSLs should be designed as an equivalent to the natural domain language for users and experts of the target domain. Final condition reflects a limited expressive capacity of DSLs and specifies that DSL is only a new way to represent some part of the whole target domain.

DSLs are always associated with some other and more general language, which is named as a base language [6]. A manner of association divides all DSLs to two classes: external and internal DSLs. The external DSLs have domain-oriented syntaxes and therefore have to be translated into commands of the base language. In contrast, internal DSLs represent a non-typical use of the base language. A scenario in this DSL contains only a subset of the features of the base language [6].

In [6] two parts of the DSL are identified: (1) a syntactic part, which defines the constructions of DSL; and (2) a semantic part, which manifests itself in the semantic model. The first part allows defining the context for working with the second one, which defines the solving procedure for the problems of the target domain. Such a structure has several advantages [5]. Firstly, DSL developers can test and develop syntax of DSL and the model independently. Secondly, many different DSLs can be designed for the same

model. Finally, developers can reuse the code of the model multiple times [7]. In such conditions, DSL represents only some additional component to the model. DSL simplifies the control procedure for the model, but it does not define it.

During design and application of DSLs developers face the challenge of syntactic analysis, which is always seems to be a resource-intensive process. Fortunately, in the case of DSL design a primitive syntactic analysis for translation DSL commands into the commands of the base language can be used. In that case compilation and correctness checking can be skipped. Kosar gives example of such DSL implementation in [16]. There are also many instruments for semi-automated developing DSLs in practice. For example, such workbench as MetaEdit [10] uses state machines for achieving this goal, which represent a native way to understand and analyze DSL syntax and its semantic model. If developers create DSL commands without a model inside, more simple analyzers cane used, such as ANTLR [11]. In our own research we selected the later tool, because we connect DSL with an external model solver. As a result, we need an instrument for translating and checking the correctness of DSL scenarios.

2.2 Constraint Satisfaction Problems

The paradigm of constraint satisfaction problems (CSPs) provides a generic method for declarative description of complex constrained or optimization problems in terms of variables and constraints [1, 2, 13–15]. Formally, CSP is a triple (V, D, C) where: $V = \{v_1, \ldots, v_n\}$ is a set of n variables, $D = \{D(v_1), \ldots, D(v_n)\}$ a corresponding set of n domains from which each variable can take its values from, and $C = \{c_1, \ldots, c_m\}$ is a set of m constraints over the values of the variables in V. Each constraint $c_i = C(V_i)$ is a logical predicate over subset of variables $V_i \subseteq V$ with an arbitrary arity k: $c_i\ (v_a, \ldots, v_k)$ that maps the Cartesian product $D(v_a) \times \ldots \times D(v_k)$ to $\{0, 1\}$. As usual the value 1 means that the value combination for $v_a, \ldots, v_k$ is allowed, and 0 otherwise. A solution for a CSP is an assignment of values for each variable in V such that all the constraints in C are satisfied.

A Distributed Constraint Satisfaction Problem (DCSP) is a CSP where the variables are distributed among agents in a Multi-Agent System and the agents are connected by relationships that represent constraints. DCSP is a suitable abstraction to solve constrained problems without global control during peer-to-peer agent communication and cooperation [16]. A DCSP can be formalized as a combination of (V, D, C, A, ∂) described as follows: V, D, C are the same as explained for an original CSP, $A = \{a_1, \ldots, a_p\}$ is a set of p agents, and ∂: $V \rightarrow A$ is a function used to map each variable v_j to its owner agent a_i. Each variable belongs to only one agent, i.e. $\forall\ v_1, \ldots, v_k \in V_i \Leftrightarrow \partial\ (v_1) = \ldots = \partial\ (v_k)$ where $V_i \subset V$ represents the subset of variables that belong to agent a_i. These subsets are distinct, i.e. $V_1 \cap \ldots \cap V_p = \varnothing$ and the union of all subsets represents the set of all variables, i.e. $V_1 \cup \ldots \cup V_p = V$. The distribution of variables among agents divides the set of constraints C into two subsets according to the variables involved within the constraint. The first set is the one of intra-agent constraints C_{intra} that represent the constraints over the variables owned by the same agent $C_{intra} = \{C(V_i) \mid \partial\ (v_1) = \ldots = \partial\ (v_k), v_1, \ldots, v_k \in V_i\}$.

The second set is the one of inter-agent constraints C_{inter} that represents the constraints over the variables owned by two or more agents. Obviously, these two subsets are distinct $C_{intra} \cap C_{inter} = \varnothing$ and complementary $C_{intra} \cup C_{inter} = C$.

The variables involved within inter-agent constraints C_{inter} are denoted as *interface variables* $V_{interface}$. Assigning values to a variable in a constraint that belongs to C_{inter} has a direct effect on all the agents, which have variables involved in the same constraint. The interface variables should take values before the rest of the variables in the system in order to satisfy the constraints inside C_{inter} firstly. Then, the satisfaction of internal constraints in C_{intra} becomes an internal problem that can be treated separately inside each agent independently of other agents. If the agent cannot find a solution for its intra-agent constraints, it fails and requests another value proposition for its interface variables. To simplify things, we will assume that there are no intra-agent constraints, i.e. $C_{intra} = \varnothing$. Therefore, all variables in V are interface variables $V = V_{interface}$.

From the functional viewpoint the most common and complete are such solvers as Choco, Gecode and Disolver [9, 12, 17]. But if the last two are closed products, Choco is an open-source product. In addition, Choco supports vectorized syntaxes for formulating the original problem that satisfies representation of constraints as a system of inequalities. Finally, this solver allows to design distributed CSPs and the solving procedure for them in terms of distributed multi-agent systems.

All mentioned solvers need a flexible mechanism to formalize the context of the problem in terms of CSP and to control the process of solution search. When traditional approaches are applied, in a case of the context changes the users have to stop the solving process, modify the source code with constructions responsible for new changes and restart the solving process. Such actions require a lot of time and reduce reactive capabilities of decision support systems. A more comfortable approach is required to modify the context of the problem dynamically, which can allow avoiding terminating the solutions search. Application of DSL seems to be reasonable for that purpose because DSL does not require restarting the solving process.

3 Proposed Framework

We propose a specific software framework, which combines description of the domain problems in terms of DSL, consequent solution of the problems by the methods of DCSP, and presentation of the results in terms of the original DSL. The framework facilitates active end-user participation in the decision process with feedback loops as it is shown in the scheme (Fig. 1). The user formulates an original problem in terms of a DSL scenario, which then is transformed in terms of the semantic model. That semantic model reformulates the input task in mathematical terms (in our case as an instance of the distributed CSP), using commands and constructions of a certain solver. When the solver obtains a solution, our framework represents that solution in the DSL terms and returns it to the user for analysis. Given the results of the analysis, the user can directly change and control the model by special DSL commands.

In our approach a DSL scenario describes not only the way of how-to-solve the problem, but also specifies the constraints which shape the solution. Syntax and semantics of DSL is designed according to the semantic model, which is the tool for the solving process in terms of our specific domain. A syntactic analyzer (or parser) is responsible for translating DSL commands into commands of the semantic model.

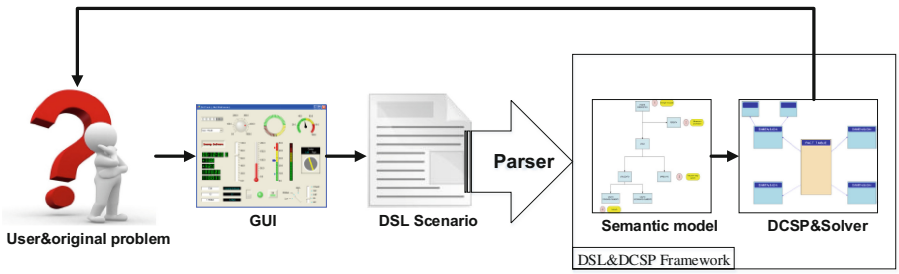

Fig. 1. The scheme of DSL processing in the framework.

As a result, the approach proposed offers an effective tool for management of the context in the semantic model. Users can concretize the context using additional DSL commands at any moment, while the model continues the solution process without needs to restart it. Furthermore, users can manage the process of solution search by filtering found answers by DSL commands. Finally, developers of decision support systems can combine developed DSL with other ways to fix the context changes in the text form. For example, they can use XML-configuration files for this purpose.

4 DSL and DCSP Application in the Railway Domain

4.1 DSL for Railway Allocation Management

The context of the railway allocation problem can change frequently because of arrivals of new trains, or changing the priority of existing services. As a result, a clear and simple way is needed to adapt new changes in terms of the DSL&DCSP framework, responsible for finding the optimal resource allocation. In the process of DSL design for the railway allocation process, it's vital to identify all the types of resources in this domain. There are three general resources for any railway station, each with specific attributes: railways, trains and service brigades.

Trains represent the main resource of the cargo railway station. Each station has different opportunities for servicing different types of trains. The amount of serviced trains defines the effectiveness of the transportation office. Each train has specific set of attributes, which help to service it by an effective and optimal manner. The whole list of these attributes includes such values as train identifier, priority, count of servicing cars, the total number of cars, types of needed services. All attributes are shown in the Fig. 2.

Railways represent the main physical place for servicing trains. Every railway is characterized by an identifier, type, total length, useful length and the list of available equipment. All attributes are shown in the Fig. 2.

Servicing brigades organize the process of servicing in terms of the railway station. The amount of brigades has to be enough to guarantee timely services for all arriving trains. Every brigade has a unique identifier, competence needed to provide services, and amount of people available for this task.

These entities constitute the basic framework of the external DSL. In addition, we need to include into DSL some helper classes to close the syntax of DSL.

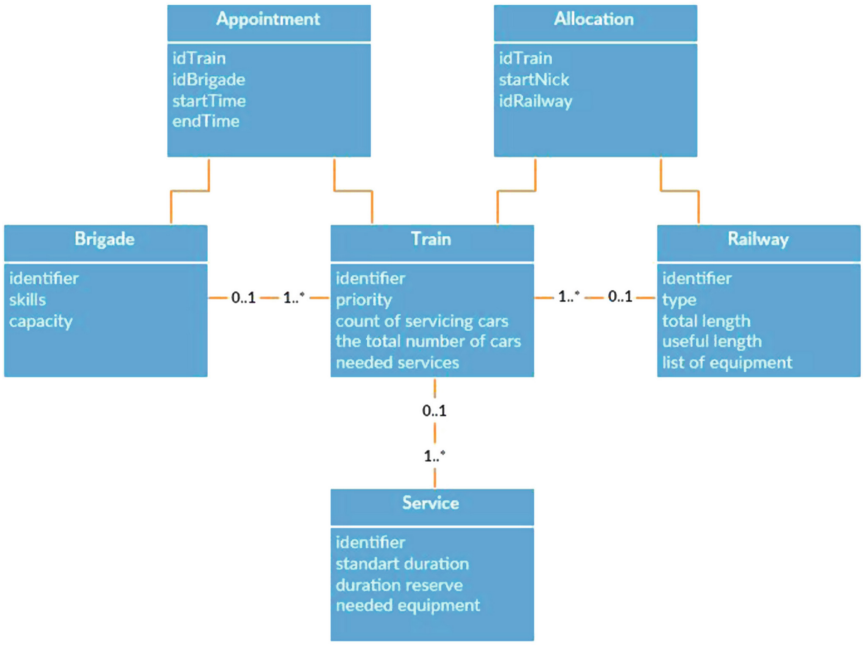

Fig. 2. The structure for DSL: The Object level

In order to work with the designed structure of main and helper classes, the next structure of DSL syntax is developed using ANTLR tool (Fig. 3). The structure of DSL consists of several blocks. The main reason for it is that syntactic analysis of such block structure becomes simple. In addition, such a structure is clear for users, who can see all the structure of the context in terms of DSL. Finally, it can help to control the structure of the model: the DSL blocks correspond to similar blocks in the semantic model and, as a result, can be naturally translated into these structures.

On Fig. 3 the meta-symbol '*' means, that a preceding sequence of objects can be written multiple times. The meta-symbols '['and ']' specify a list of single-typed entities.

The structure of objects above organizes only the first level of DSL – the level of the objects. In addition, we have to develop the level of functions to manage the objects and their context. In our case this functional level has the next structure (Fig. 4).

Using these commands, we can specify the context for the objects and their management. In addition, such a command structure allows to simplify and accelerate the solving procedure with additional details needed for finding the optimal solutions.

When all objects and functions are designed in terms of the DSL, such DSL scenario is translated into the commands of the semantic model using a syntactic analyzer. In this case we have no opportunities to intervene in the process of solutions search directly. We can only manage it by DSL commands. When the solver found the optimal solution, it translates the solution to the DSL terms using a syntactic analyzer. And then this solution can be checked and filtered by the end user. The main advantage in this case is that the final solution represents not a mathematical view, but a

```
Trains
({id type priority length timeArrival timeDeparture timeReserve
[services] wagonsToService})*
EndTrainsBlock;

Services
   ({name priority mayBeInParallel [skills]
standartDuration timeReserve [equipment]})*
endServiceBlock;

Brigades  ({id [ skills ] capacity})* endBrigadeBlock;

Railways
   ({id type totalLength usefulLength [equipment]})*
endRailwayBlock;

Appointments
   ({idBrigade->idTrain timeStart timeEnd})*
endAppointmentBlock;

Allocation
   ({idTrain->idRailway startNick
timeStartOccupation timeEndOccupation})*
endAllocationBlock;
```

Fig. 3. DSL objects

```
Relocate idTrain [from idOldRailway] to idNewRailway;
```
This command allows to move Train with identifier idTrain from railway with identifier idOldRailway to railway idNewRailway.

```
Move forward/back idTrain by countShifts;
```
This command allows to move Train within current railway forward or back by the count of shifts equals to countShifts.

```
Put idTrain on newStartNick;
```
This command places the train with identifier idTrain on start nick with number newStartNick.

```
ChangePriority idTrain/nameService on newPriority;
```
This command allows to change priority of train or service on some new value newPriority

```
Appoint idBrigade on idTrain from startTime [to endTime] perform
nameService;
```
This command allows to appoint brigade with identifier idBrigade for servicing nameService on Train idTrain within the period from startTime to endTime.

```
Get info on brigade/train/service/railway idEntity;
```
This command returns all the attributes for brigade, train, service or railway identified by idEntity.

Fig. 4. DSL functions

domain-oriented description, which contents is clear for the user. In these circumstances we might not to spend time on transforming the solution found in terms of the target domain, but check it immediately and use in practice.

Table 1. Description of model's parameters

Parameter	Description
$1..M$	identifiers of arriving trains
$1..K$	identifiers of railways
L_j	length of j-railway
l_i	length of i-train
N_i	amount of cars for servicing i-train
$Tarr_i$	time arrival i-train
Tsw_i	start time of servicing i-train
Top_i	time for servicing i-train
W_i	identificator of railway for i-train
$C_{i,0}$	start nick i-train
$\delta_{i,t}$	size of shift i-train at the time moment t
T_{cur}	current time moment
$Tshift_{i,t}$	time of shift i-train at the time moment t

4.2 Mathematical Model for Railway Allocation

To formalize the mathematic model of DCSP for solving the railway allocation problem, firstly identify all the set of parameters, based on the above description of entities (Sect. 4.1) (Table 1).

These parameters are needed for solving the railway transportation problem effectively and are not redundant. They have to satisfy the system of constraints (1). The first six constraints are needed to distribute arriving trains in time and protect them against time-conflicts. These constraints assure that trains are serviced in time and no longer and have no conflicts within the timetable. Other constraints control mutual arrangement of trains in space of railways, block trains under the service for shifting.

$$\begin{cases} t \in [0; T_{\max}] \\ Tcur \in [0; T_{\max}] \\ Top_i \geq 0 \\ Tarr_i \in [0, T_{\max}], i = 1..M \\ Tsw_i \in [0, T_{\max}], i = 1..M \\ Tsw_i \geq Tarr_i, i = 1..M \\ 1 \leq W_i \leq K, i = 1..M \\ 0 \leq C_{i,0} \leq L_{W_i}, i = 1..M \\ \sum_{i:W_i=j} l_i \leq , L_j, i = 1..M, j = 1..K \\ \delta_{i,t} = -\infty, t < Tarr_i, i = 1..M \\ \delta_{i,v}, = 0, i = 1..M, v = Tarr_i Tsw_i \leq v \leq Tsw_i + Top_i \\ 0 \leq C_{i,0} + \sum_{t=0}^{Tcur} \delta_{i,t} \leq L_{W_i}, i = 1..M \\ 0 \leq C_{i,0} + \sum_{t=0}^{Tcur} \delta_{i,t} + l_i \leq L_{W_i}, i = 1..M \\ C_{i,0} + \sum_{t=0}^{Tcur} \delta_{it} + l_i \leq C_{i+1,0} + \sum_{t=0}^{Tcur} \delta_{i+1,t}, i = 1..(M-1) \end{cases} \quad (1)$$

When constrains are formulated, we can identify the goal function for solving procedure. In our case the goal is represented by the next system.

$$\begin{cases} \sum_{i=1}^{M} N_i \rightarrow max \\ \sum_{i=1}^{M} \sum_{t=0}^{Tcur} Tshift_{i,t} \rightarrow min \end{cases} \quad (2)$$

The goal function maximizes the amount of serviced trains, but minimizes the time for their shifting within the operations, which is useless and decreases the effectiveness of transportation.

Given the mathematical model, translation of this model to the terms of Choco solver produces two vectors – for the left and right sides of all constraints and identifying the relations between them, using operators "<, >, =, !=". Then the target function is represented in terms of Choco solver. At the final stage the vectorized view of Choco results is translated to DSL.

4.3 Subject-Oriented GUI

After design of all the constructions for DSL and its DCSP-based mathematical model, we can think over the design of GUI, to simplify the procedure of creating DSL scenarios. First of all, we have to understand, that developing GUI must contain only elements, equivalent to DSL components. No more elements are needed, because no more elements are supported by DSL and, as a result, by the model. In this context GUI should be a graphical equivalent to DSL constructions and therefore is subject-oriented.

The second idea for implementation of GUI is that developed DSL is oriented on the procedure of configuration contexts. As a result, it can be useful to define a first interface element within the main frame of GUI as the working area which reflects the current context status. The second interface area, the panel of objects, contains available objects for defining the context. The GUI prototype is demonstrated in Fig. 5.

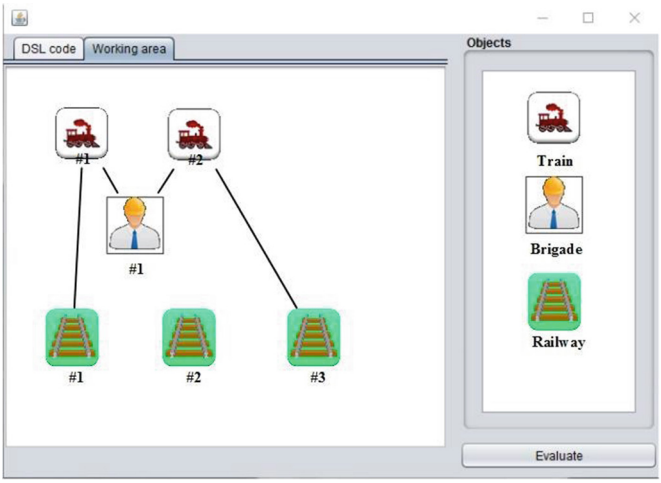

Fig. 5. The GUI prototype

When a user wants to change the context, the user drags the needed element from the panel of objects onto the working area. Then the user has to fill all the attributes for the dragged element in a specific dialog. After the element created, it is added to the context. To include new added elements into the solving process, the button "Evaluate" have to be pressed to start the procedure of finding solutions. If solutions are found, GUI shows them with connections between the objects inside the working area.

To improve the GUI, time scale can be added into the working area. Along this line all the objects are shown at every time of a moment. Such improvement can allow to control the workload for objects at any moment. In addition, we can use this scheme to create a timetable for resource allocation in future outlooks.

5 Evaluation

Let demonstrate using of the DSL&DCSP framework proposed in a real case of the railway resource allocation. Suppose, for servicing trains the station has the next amount of railways (Table 2).

Suppose that three trains arrive at the station, and they have the following characteristics (Table 3).

In terms of DSL we have the following description of that context (Fig. 6).

Table 2. Characteristics of available railways.

Railway identifier	1	2	3
Type	Universal	Universal	Universal
Total length (in cars)	30	24	10
Useful length	28	22	7
Available equipment	Transporter, forklift, conveyor	Transporter, forklift, conveyor	Transporter, forklift

Table 3. Characteristics of arriving trains.

Train identifier	1	2	3
Type	dry cargo	dry cargo	tanks
Priority	High	High	regular
Total length (in cars)	14	16	8
Time arrival	14:03	14:47	15:12
Time departure	16:11	16:19	17:49
Needed services	Inspection, loading, unloading	Inspection, loading, unloading	Inspection, initial service
Cars to service	9	6	1

```
Services
1 1 0,67 0
21 0,450,05 [ 1 3 ]
31 10,5 [ 1 3 ]
endServicesBlock;
Equipment
1 Transporter
2 Forklift
3 Conveyor
endEquipmentBlock;
Railways
1 1 30 28 [ 1 2 3 ]
1 1 24 22 [ 1 2 3 ]
1 1 10 7 [ 1 2 ]
endRailwayBlock;
Trains
1 1 1 14 14:03 16:11 0 [ 1 3 4 ] 9
2 1 1 16 14:47 16:19 1 [ 1 3 4 ] 6
3 2 2 8 15:12 17:49 0 [ 1 2 ] 1
EndTrainsBlock;
```

Fig. 6. DSL scenario (objects)

To simplify the task, we can suppose, that four universal Brigades are available for servicing tasks. Then DSL&DCSP framework translates this DSL specification to the semantic model, which is based on the model of constraint satisfaction (Fig. 7).

Inside the DSL&DCSP framework the Choco solver accepts the constraints model and computes the result in terms of that model. During the final step, the Choco result is translated to the equivalent DSL-based form. The user obtains the result in the following form (Fig. 8).

As we can see, this solution is optimal for our task: services are available for parallel working and the amount of railways is enough for parallel servicing. And finally, trains with identifiers 2 and 3 are placed along the useful length of the second railway and can be located there. The framework provides an optimal solution in terms of effective using of available length of the railways and the factor of service brigades using. Besides, the final solution has ready-to-use domain commands for trains and brigade's allocation. It creates the stable context for future railway resources allocation and allows not only save reserves for changes in the allocation context, but increase the beneficial use of all types the resources.

In comparison, a currently used system, exploited in the railway transportation office, proposes another solution. That system allocates the train 3 on the railway 3 without combination with the train 2. This solution is not optimal, because it does not create a space reserve for the future outlook. Moreover, the currently used system spends more time solving the problem. The statistics is shown in Table 4.

Data in Table 5 show that our proposed DSL&DCSP framework spends on average 2/3 times less than the currently used system. It means that the developed framework is more flexible and is more oriented on the context.

```
Solver solver = new Solver();
IntVar L = VariableFactory.fixed("L", 30, solver);
IntVar deltMin = VariableFactory.fixed("deltMin", 20, solver);
IntVar[] l = VariableFactory.enumerated("l", 10, L.getValue(), solver);
IntVar[] s = VariableFactory.enumerated("s", 10, L.getValue(), solver);
IntVar[] delts = VF.enumerated ("delts", 9, 0, L.getValue(), solver);
solver.post(IntConstraintFactory.sum(new IntVar[]{s,l,delts}, "<=", L));
solver.post(IntConstraintFactory.sum(new
IntVar[]{s,l,VariableFactory.minus(delts)}, "<", s[0]));
```

Fig. 7. Choco DCSP-model view

```
Relocate 1 to 1 from 14:03;
Relocate 2 to 2 from 14:47;
Relocate 3 to 2 from 15:12;
Appoint 1 on 1 from 14:03 perform 1;
Appoint 1 on 1 from 14:03 perform 3;
Appoint 1 on 1 from 14:43 perform 4;
Appoint 2 on 2 from 14:47 perform 1;
Appoint 2 on 3 from 15:17 perform 1;
Appoint 3 on 2 from 14:47 perform 3;
Appoint 3 on 2 from 15:27 perform 4;
Appoint 4 on from 15:17 perform 2;
```

Fig. 8. DSL representation of the solution

Table 4. The comparison statistics

Model	Proposed DSL&DCSP framework	Current system
Time for finding solution (sec.)	17	39
Is solution optimal?	Yes	No

Table 5. Comparison of an existing system and DSL&DCSP Framework: average time for finding the solution (measured in sec.) The results of DSL&DCSP Framework are marked by bold.

		Amount of trains			
		10	25	50	100
Time Reserve (h.)	1	1.96 (**1.78**)	4.66 (**3.33**)	7.29 (**6.08**)	28.61 (**21.84**)
	2	10.38 (**7.41**)	16.23 (**12.48**)	23.47 (**16.77**)	160.27 (**126.19**)
	3	20.93 (**17.44**)	60.60 (**35.64**)	174.38 (**116.25**)	280.19 (**200.13**)

In addition to the time of finding solution, another additional characteristic is used for assessing the quality of the found solution – the utilization rate of a useful length of railways. It shows the quality of using railways and allows to evaluate the load of every railway: the higher the value, the better the quality of the distribution between the railways.

6 Conclusion

In our research we explored an opportunity to organize and simplify decision support during complex resource allocation processes by combining domain specific languages and distributed constraint satisfaction techniques. In the result we have developed DSL&DCSP Framework for decision support in the railway allocation process. That framework includes a domain-oriented external DSL for continuous accounting of changes in the domain, a syntactic analyzer for translating DSL commands into the terms of the semantic model and the DCSP solver Choco inside for solution search. The external DSL uses the concepts, which are equivalent to the main resources of the railway station: trains, railways, brigades and services. The framework translates these concepts into the terms of DCSP, which uses the Choco syntax. We also propose a prototype of GUI, which replaces the textual representation.

In contrast to other examples of using DSL&DCSP inside one framework (like [18]), our solution demonstrates the DSL&DCSP application for one specific domain of railway allocation problem. It facilitates effective specification of varying instances of the same model via different DSL scenarios.

Results of experiments show that the framework solves the allocation problem faster, then an existing system. In addition, the framework facilitates easy user-driven changes of specifications of the underlying model, because the framework uses GUI for this goal instead of the command line interface of the existing system.

According to theory and the case, shown above, we can say that proposed combination of DSL and DCSP becomes a really effective and clear tool for managing the dynamic context in the complex problem solving process. Moreover, we can approve, that proposed DSL&DCSP framework can be extended and applied to other problems. It is vital to outline, that such extension requires only changes in the mechanism of translating an external DSL into the base language. Other parts (a DCSP model, the solving procedure etc.) can be generated automatically. As a result, partial independence from the domain specifics makes the framework flexible and adaptable to different domains.

Playing the central role for the user-oriented context management, designed DSL gives many attractive opportunities for end-users. First of all, our DSL determines the context as well as concretizes it by special commands and objects. DSL also facilitates direct and comprehensive communication between developers and experts in the specific domain. In addition, DSL saves time of decision makers - understanding commands for the context management becomes easy because of their mnemonic nature.

Finally fused with the DCSP-based semantic model DSL offers new features in development new methods for finding solutions in dynamics contexts by providing a flexible mechanism for direct controlling the effectiveness and correctness of the model and its attributes by domain experts.

Our results show that DSL does not require redesign of existing models for solving problems. It can be integrated into current solving methods using XML files, containing information needed for DSL and for the next translation in terms of its semantic

model. Such XML files can be used and as an equivalent to DSL, because of their nature, similar to the real language, its semantics and syntax.

All these findings support our initial hypothesis about an important role of DSLs in decision-making process due to significant improving the quality of the whole system and increasing its manageability and flexibility.

Planning further research, we consider such an extension of the proposed DSL&DCSP framework, which provides not only the optimal allocation of trains within one station, but also supports communication of different stakeholders during a complex decision process within different stations, distant from each other. In this case we will face the problem of semantic heterogeneity. We expect that new solutions should be proposed for real-time unification of disparate and heterogeneous DSLs into one single language, allowing precise specification of the interests and preferences of different stakeholders. In this context, we have not to mechanically combine parts of different DSLs, but find the way for communication of different stakeholders with opposite views on the context and its influence on them.

For achieving this goal an approach, described by Pereira, can be used. In [19] Pereira et al. used ontologies and specific grammars to create DSL structures. Starting with ontological specification of the target domain and using automated analyzers for parsing, they end with the DSL scenario, derived from fragmented descriptions of the problem. This approach requires changes in its structure, represented by an ontological analyzer, but allows combine different specifications of the current domain in final DSL language.

References

1. Binmore, K.: Rational decisions. Princeton University Press, Princeton (2009)
2. Hodgson, M.: On the Limits of Rational Choice Theory. Econ. Thought **1**, 94–108 (2012)
3. Fernandez-Lopez, M., Gomez-Perez, A.: Overview and analysis of methodologies for building ontologies. Knowl. Eng. Rev. **17**(2), 129–156 (2002). Cambridge University Press
4. Shcherbina, O.: Nonserial dynamic programming and tree decomposition in discrete optimization. In: Waldmann, K.-H., Stocker, U.M. (eds.) Proceedings of International Conference on Operations Research, pp. 155–160. Springer, Berlin (2007)
5. Martin, F.: Domain Specific Languages. Addison Wesley, Upper Saddle River (2010)
6. Terence, P.: Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages. Pragmatic Bookshelf, Frisco (2012)
7. Eric, E.: Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley (2013)
8. Makoto, Y.: Distributed Constraint Satisfaction. Springer, Heidelberg (2001)
9. Choco solver. http://choco-solver.org/
10. MetaCase+. http://www.metacase.com/
11. ANother Tool for Language Recognition (ANTLR). http://www.antlr.org/
12. Gecode toolkit. http://www.gecode.org/
13. Barták, R. Constraint programming: in pursuit of the holy grail. In: Proceedings of WDS 1999 (Invited Lecture), pp. 555–564 (1999)
14. Eisenberg, C.: Distributed Constraint Satisfaction for Coordinating and Integrating a Large-Scale, Heterogeneous Enterprise, University of London (2003)

15. Bacchus, F., van Beek, P.: On the conversion between non-binary and binary constraint satisfaction problems. In: Proceedings of the 15th National Conference on Artificial Intelligence (AAAI 1998) and of the 10th Conference on Innovative Applications of Artificial Intelligence (IAAI 1998), pp. 311–318 (1998)
16. Kosar, T., Martınez Lopez, P., Barrientos, P., Mernik, M. A preliminary study on various implementation approaches of domain-specific language. In: Information and Software Technology, pp. 390–405. Elsevier (2008)
17. Disolver. http://research.microsoft.com/apps/pubs/default.aspx?id=64335
18. Prud'homme, C., Lorca, X., Douence, R., Jussien, N.: Propagation engine prototyping with a domain specific language. Constraints **19**(1), 57–77 (2013)
19. Pereira, M., Fonseca, J., Henriques, P.: Ontological approach for DSL development. In: Computer Languages, Systems & Structures, pp. 35–52. Elsevier (2016)