Using DSLs to help people solve rule-based problems
Draft Paper

Nico Naus[1] and Johan Jeuring[1,2]

[1]Utrecht University
[2]Faculty of Management, Science and Technology, Open University of the Netherlands

Abstract. We present a DSL that relates and unifies software supporting rule-based problem solving from different domains. This DSL is used to assist users when they attempt to solve a problem. We present three examples frameworks for problem solving to motivate our approach: the ideas framework, PuzzleScript and iTasks. We then present our DSL, together with four general search algorithms that can be used to solve some of these problems.

1 Introduction

Many software frameworks and systems support, model, or automate the process of human problem solving. With a problem we mean anything from a game or a puzzle to solving an exercise in physics or another subject, or search and rescue people in need at sea. Typical examples of systems supporting problem solving are workflow management systems, intelligent tutoring systems, and expert systems.

A user of a system supporting problem solving sometimes needs help in making a decision or taking a step towards a particular goal. In the case of a game or a puzzle, a user might get stuck, and need a step in the right direction. For supporting a student solving an exercise in an intelligent tutoring systems, hints are essential [8]. In search and rescue systems, hints can quickly give insight in the current situation, and can help a user in understanding why a next step has to be taken. A user has to take a decision under pressure of time and potential danger. Automatically suggesting and explaining the best option to perform may reduce the chance of human error.

In all of the above examples a user follows a, potentially very flexible, process, and needs information about where she is in the process, where she should go next, and why she should go there. This paper proposes a unified framework to describe processes for problem solving. For this purpose, we use a domain specific language (DSL). The steps necessary for solving a problem follow a strategy, and the strategy language we developed in earlier research [1] is a good candidate for describing the problem solving process. Different systems use

problem-solving processes desribed in this DSL in different ways. Giving a hint in an intelligent tutoring system for solving equations often amounts to returning the next steps prescribed by the solving procedure, where providing a hint for a more complicated problem such as the travelling salesman problem requires more involved techniques. We obtain the different instances of problem solving processes by implementing our DSL in different ways. Thus we have a unified framework for describing problem solving processes, which can be instantiated for different purposes by selecting a different implementation.

This paper is organized as follows. Section 2 discusses some examples for which problem-solving assistance is desirable. Section 3 introduces a DSL for describing the rule-based problem solving processes. Section 4 presents several methods for solving the various problems. We classify these methods in Section 5. Section 6 concludes.

2 Examples

This section illustrates and motivates our goal of providing help to people using a rule-based problem solving system by giving three examples: the ideas framework [1], PuzzleScript [4], and the iTasks system [6]. Each of these frameworks can describe a variety of problems. We briefly introduce each framework, show an example problem described in the framework, and explain what kind of problem solving assistance is desired.

2.1 ideas

The ideas framework is used to develop services to support users when stepwise solving exercises in an intelligent tutoring system for a domain like mathematics or logic. It is a general framework used to construct the expert knowledge of an intelligent tutoring system (ITS). The framework has been applied in the domains of mathematics [1], programming [2] and communication skills [3].

$$
\begin{array}{lcll}
\mathit{dnfStrategy} & = & \mathit{label}\ \texttt{"Constants"} & (\mathit{repeat}\ (\mathit{topDown}\ \mathit{constants})) \\
 & <*> & \mathit{label}\ \texttt{"Definitions"} & (\mathit{repeat}\ (\mathit{bottomUp}\ \mathit{definitions})) \\
 & <*> & \mathit{label}\ \texttt{"Negations"} & (\mathit{repeat}\ (\mathit{topDown}\ \mathit{negations})) \\
 & <*> & \mathit{label}\ \texttt{"Distribution"} & (\mathit{repeat}\ (\mathit{somewhere}\ \mathit{distribution}))
\end{array}
$$

Fig. 1: A problem solving strategy, formulated in terms of rules and combinators

The central component of the expert knowledge for an ITS is expressed as a so-called *strategy* in ideas. For example, Figure 1 gives part of a strategy for the problem of rewriting a logic expression to disjunctive normal form (for the complete strategy see Heeren et al. [1]). The framework takes this strategy

to offer various services, among which a service that diagnoses a step from a student, and a service that gives a next step to solve a problem. A student receives a logic expression, and stepwise rewrites this expression to disjunctive normal form using services based on the above strategy. At each step, a student can request a hint, or ask for feedback on her current expression. If no rules can be applied anymore, the expression is in normal form, and the 'ready' service reports that.

dnfStrategy describes a rule-based process that solves the problem of converting an expression to disjunctive normal form. It is expressed in terms of combinators like $< * >$, *repeat* and *somewhere*, and further substrategies. Additionally, some parts have been labeled with information that can be used in feedback.

2.2 PuzzleScript

PuzzleScript is an open source HTML5 Puzzle Game Engine [4]. It supports easy puzzle scripting. Its central component is a DSL for describing a game. PuzzleScript compiles a puzzle described in this DSL into an HTML5 puzzle game. Using the DSL, we describe a puzzle as a list of objects, rules that define the behavior of the game, a win condition, collision information and one or more levels.

```
======== ======
OBJECTS  RULES
======== ======
Background    [ >Player | Crate ] -> [ > Player | >Crate ]
Green
Target        ==============
DarkBlue      COLLISIONLAYERS
              ==============    ===========
Wall                            LEVELS          =======
Brown         Background        ===========     LEGEND
              Target            #########       =======
Player        Player, Wall, Crate
Blue                            #.......#       . = Background
                                #.....@.#       # = Wall
              ============      #.P.*.O.#       P = Player
Crate         WINCONDITIONS     #.......#       * = Crate
Orange        ============      #.......#       @ = Crate and Target
              All Crate on Target #########     O = Target
```

Fig. 2: Partial definition of the Hello-world example of PuzzleScript

The standard hello-world example for PuzzleScript is given in Figure 2. It describes a simple crate-pusher game, also called Sokoban. Objects are back-

ground, walls, crates, the player and the targets for the crates. There is just a single rule, which describes that if a player moves into a crate, the crate moves with the player. Objects appearing on the same line in the collision layers are not allowed to pass trough each other. The winning condition is reached when all the targets have a crate on them. Finally, a start-level is specified under *LEVELS*.

In a difficult game, we might want to offer hints on how to proceed to a player. Based on the state of the game, the *RULES*, *COLLISIONLAYERS* and *WINCONDITIONS*, we can calculate a hint for a user [5]. This same information can also be used to check if a game still can be solved in the current state. For example, if a crate gets stuck in a corner, the game cannot be solved anymore.

2.3 iTasks

iTasks [6] supports task-oriented programming in the pure functional programming language Clean. Clean is very similar to Haskell, with a few exceptions. A data declaration starts with ::, types of function arguments are not separated by a function arrow (→) but by a space, and class contexts are written at the end of a type, starting with a |.

```
:: GameState = { board :: [Int], dim :: Int, ePos :: Int }
:: Dir = East | West | North | South
boardStore :: Shared GameState
boardStore = sharedStore "game" { board = [4, 8, 6, 1, 7, 0, 2, 5, 3], dim = 3, ePos = 5 }
slidePuzzle :: Task GameState
slidePuzzle =
        viewSharedInformation "Sliding Tile Puzzle" [ViewWith viewBoard] boardStore
    >>* map (λdir → OnAction (Action (toString dir) [])
                             (ifValue (checkStep dir)
                               (λst → set (applyStep dir st) boardStore >>| slidePuzzle)
                             )
            )
            [West, East, North, South]
viewBoard :: GameState      → HtmlTag
checkStep :: Dir GameState → Bool
applyStep :: Dir GameState → GameState
```

Fig. 3: Example iTasks program, formulated by composing tasks

Figure 3 gives the (partial) source code of an iTasks program for a tile sliding puzzle, as shown in Figure 4. In this puzzle, the player arranges all tiles in order, by using the empty position to slide the tiles over the board.

The record type *GameState* holds the board configuration, the dimension of the puzzle, and the position of the empty slot. *Dir* defines the kind of moves

(a) Initial state (b) Goal state

Fig. 4: Instance of a block sliding puzzle, of dimension 3 x 3

a player can perform. *slidePuzzle* implements the puzzle. It is composed out of base tasks, task combinators, and standard Clean functions. A task is a monadic structure. Its evaluation is driven by events and handling an effect has an potentially changes a shared state.

slidePuzzle uses the standard task for viewing information, stored in the Shared Data Sources [6] (SDS), to display the current state. Then, it uses the step combinator $\gg\star$ to combine the viewing task with the tasks offering the possible options. The step combinator performs several functions. First, it allows us to put two tasks in sequence. The second task gets the result from the first task. The second task consists of a list of *TaskSteps*, one of wich has to be chosen. A *TaskStep* is a regular task combined with an action to trigger it, and a condition to enable the trigger. We use a *map* to generate the four options a player can choose between.

When running *slidePuzzle*, the iTasks system will render the *boardStore*, together with four buttons to move the tile. If an action is not applicable (for example, a move *North* when the open slot is already at the top of the game), the button is disabled. When a user clicks on an enabled button, the tile is moved accordingly.

The goal of the puzzle is to move all tiles in positions so that they appear in order, as shown in Figure 4b. We want to extend the functionality of our program to give a user a hint if she does not know how to proceed. We could implement this in an ad-hoc fashion by developing hint functionality for each iTask program on which we want it. Alternatively, we will use the same framework as we want to use in the previous examples, giving jint functionality for free in every iTask program.

3 Problem Formalization

Russell and Norvig [7] define a well-defined problem as follows:

Initial state The state of the problem that you want to solve
Operator set The set of steps that can be taken, together with their effects
Goal test A predicate that is True if the problem is solved

Path Cost function A function that describes the cost of each operation

We use a slightly simplified definition of a problem. In our case, we have an initial state, represented by a value of type a, and operator set, what we call the problem solving strategy (strategy in short), represented by the type *RTree*, and a goal test, represented by the predicate *Goal*.

Figure 5 gives the types of the components of our DSL.

```
state     :: a
Rule      :: Rule Name Effect
Name      :: String
Effect    :: a → a
Goal      :: Predicate
Predicate :: a → Bool

:: RTree = Seq [RTree]
         | Choice [RTree]
         | Parallel [RTree]
         | Condition Predicate RTree
         | Rule Rule
         | Empty
```

Fig. 5: Types of the components of our DSL

The *Goal* can be reached by performing one or more rules after eachother to end up in a node of the *RTree* where the *Goal* condition is met. We consider this point in the *RTree* to be the solution. The depth of this point is the solution depth d.

3.1 Sliding puzzle in our DSL

Figure 6 gives the example iTask program from Figure 4 in our DSL format.

4 Solving

The DSL introduced in the previous section offers a uniform approach to describe rule-based problems. However, different classes of problems require different approaches to solving such problems. This section describes how we can view the DSL as an interface, for which we can provide different implementations to obtain different ways to approach a problem, and to obtain various services, in particular for providing hints.

This section shows implementations of a service for giving hints for different classes of problems. All implementations take the strategy *RTree*, the current state in the form of a value of a type a and the goal test *Goal*, and return a list of zero or more hints (rules) that can be taken at this point in solving the problem. As will become clear in the coming sections, not all implementations require a goal test. Some implementations requre an additional scoring function like fitness or a heuristic to complete them. We also attempt to state what guarantees can be given for the implementation. Will the implementation find a solution, and if so, what is the quality of the found solution?

The implementations we give in this section all use Clean syntax.

```
slidePuzzle :: RTree
slidePuzzle =
  Seq [Rule "View information" id
      , Choice [Condition (checkStep Up)    (Seq [Rule "Move up"    (applyStep Up)
                                                 , slidePuzzle])
                , Condition (checkStep Down) (Seq [Rule "Move down"  (applyStep Down)
                                                 , slidePuzzle])
                , Condition (checkStep Left)  (Seq [Rule "Move left"  (applyStep Left)
                                                 , slidePuzzle])
                , Condition (checkStep Right) (Seq [Rule "Move right" (applyStep Right)
                                                 , slidePuzzle])
                ]
       ]
checkStep :: Dir GameState → Bool
applyStep :: Dir GameState → GameState
state :: GameState
goalTest :: GameState → Bool
goalTest {board, dim} = [0..((dim * dim) − 1)] ≡ board
```

Fig. 6: Sliding puzzle in our DSL

4.1 allFirsts

allFirsts returns the first steps that can be taken given a state and an *RTree* value. Since it does no take the goal into account, these steps are only relevant for problem domains where it is possible to precisely describe the next step to be taken in a solution towards a goal using an *RTree* value. Examples of such domains are all kinds of mathematical and logic exercises. The *allFirsts* service is used if the tree describes only the steps that are always on a path to the goal. That is the case in for example the ideas framework.

For this algorithm, we cannot give any guarantee about the given hint. Since it simply returns all steps a user can take, it is completely up to the programmer to guarantee that only correct steps can be taken.

4.2 Fitness

Just as *allFirsts*, *fitnessHint* only looks at the first next steps specified by the *RTree*. From these next steps, it selects the best by calculating the fitness of each step, and taking the step with maximum fitness. It assumes that the fitness function ensures that a user gets closer to the goal if she follows the hints, but of course this depends on the relation between the fitness function and the goal. If a programmer passes a 'true' fitness function as argument, that is, a step selected by the fitness function brings a user closer to the goal and not to a local optimum, then the step returned by *fitnessHint* is part of a sequence of steps that leads to the goal.

```
allFirsts      :: (RTree a) a → [String]
allFirsts t d = map toName (topRules d t)

topRules :: a (RTree a      ) → [Rule a]
topRules   _ Empty          = []
topRules   _ (Leaf r        ) = [r]
topRules   _ (Seq []        ) = []
topRules   d (Seq [t : ts]  ) = case topRules d t of
                                  [] = topRules d (Seq ts)
                                  x = x
topRules   d (Choice t      ) = flatten (map (topRules d) t)
topRules   d (Parallel t    ) = flatten (map (topRules d) t)
topRules   d (Condition c t) |  c d = topRules d t
                             |  otherwise = []

toName :: (Rule a)   → String
toName   (Rule n _) = n
```

Fig. 7: Definition of the *allFirsts* hint service

```
fitnessHint :: (a → Int) a (RTree a) → [String]
fitnessHint f              d t       =
  map toName (findBest (map (λn → (n, score f d n)) (topRules d t)))

findBest :: [(Rule a, Int)] → [Rule a]
findBest   []               = []
findBest   [(t1, n) : xs]   = fst (helper1 ([t1], n) xs)

helper1 :: ([Rule a], Int) [(Rule a, Int)] → ([Rule a], Int)
helper1   a                []              = a
helper1   (tl, n)          [(t2, n2) : xs] |  n ≡ n2 = helper1 (tl ++ [t2], n) xs
                                            |  n < n2 = helper1 (tl, n) xs
                                            |  n > n2 = helper1 ([t2], n2) xs

score :: (a → Int) a (Rule a) → Int
score f            d (Rule _ e) = f (e d)
```

Fig. 8: Definition of the fitness-based hint service

4.3 Brute force

Figure 9 gives a brute force algorithm *bfHint*. It takes a goal, a state and an *RTree*, and returns a list of first steps that can be taken. It uses the function *bfTrace*, which returns all paths that reach the goal. Since we only need the first step to produce a hint, we map the *hd* function over it. *bfTrace* works recursively. It first takes the previous expansion, and filters out the uncompleted once, and keeps the traces of the successful expansions. If there are one or more successful expansions, we return these traces. If no expansions are successful, all states are expanded and *bfTrace* is called recursively. This means that we search breadth first. If multiple solutions are found at the same depth, all these solutions are returned. Therefore, the return type of *bfHint* is a list of *String*.

If *bfHint* returns a hint, then by definition this is a step in a sequence of steps that achieves the goal.

```
bfHint :: (a → Bool) a          (RTree a) → [String]
bFHint  done         domain tree          = map hd (bfTrace done [([], tree, domain)])
bfTrace :: (a → Bool) [([String], RTree a, a)] → [[String]]
bfTrace  done         []      = []
bfTrace  done         items =
  case [h \\ (h, _, d) ← items | done d] of
    [] = bfTrace done (flatten (map expand items))
    x  = x
expand :: ([String], TTree a, a) → [([String], TTree a, a)]
```

Fig. 9: Brute force algorithm

We have only listed the type of *expand* due to a lack of space.

4.4 Heuristic search

A potential problem with the brute force algorithm is that it expands every state until a state fulfills the goal predicate. This might be computationally very expensive. We can try to reach the goal using fewer resources by using a heuristic. A heuristic is a function $hr :: a \to Int$, with $hr \equiv 0$ when we reach our goal. This heuristic function is used in the search algorithm to search for a solution in a more informed way. A heuristic function differs from a fitness function in the sense that we drop the requirement that the function may not lead to a local optimum. The implementation takes this into account and backtracks if it gets stuck in a local optimum.

Figure 10 gives our *heuristicHint* algorithm. *heuristicHint* initializes the arguments for *hStep* and returns the head of the trace, once a solution is found. *hStep* expands the first tuple in the list. There is either just one element, or the

```
heuristicHint :: (a → Int) (a → Bool) (RTree a) a     → String
heuristicHint   hr         done       t           dom = hd (hStep hr done [(0, ([], t, dom))])
hStep :: (a → Int)    (a → Bool) [(Int, ([String], RTree a, a))] → [String]
hStep   _             _          []                              = []
hStep   hr            done       [(_, t) : xs]                   =
  hDecide hr done (sortScore ((map (λ(h, t, d) → (hr d, (h, t, d))) (expand t)) ++ xs))
hDecide :: (a → Int) (a → Bool) [(Int, ([String], RTree a, a))] → [String]
hDecide   hr         done       [(_, (h, _, d)) : xns] | done d = h
hDecide   hr         done       [(_, (h, Empty, d)) : xns]      = hDecide hr done xns
hDecide   hr         done       [(_, (h, t, d)) : xns]          = hStep hr done xns
```

Fig. 10: The *heuristicHint* algorithm

elements are sorted by the previous iteration, and the first item has the best score. After expansion, all new states are scored using the heuristic function, and then combined with the remaining input. *hDecide* looks for an expansion that fulfills the goal condition, removes fully expanded states, and if no solution is present, starts the next iteration of the expansion.

4.5 Other algorithms

The algorithms described in this section have been implemented in our framework. There are many other algorithms that support solving problems of the kind described in Section 3. A programmer can implement these once, and then solve problems using our framework. Some common algorithms not listed above are A*, Hill climbing, and probabilistic annealing.

5 Classification

This section classifies the algorithms given in Section 4. As mentioned before, different problems call for different algorithms. Table 1 gives a compact overview of some of the properties of the approaches. The first column lists whether or not an algorithm uses additional information specific to the problem. In the second column, look-ahead, we mark how far an algorithm has to look ahead in order to calculate a hint. A lookahead of one means that the algorithm inspects the *RTree* just one level deep, a d stands for the solution depth. The last column lists the complexity of the different algorithms. Here, b stands for the branching factor of the *RTree*.

6 Conclusions

In this draft paper, we have presented a DSL that unifies and relates rule-based problems in different domains. In the full paper, we will validate our DSL. We will

Algorithm	Informed	Look-ahead	Complexity
allFirsts	✗	1	1
fitnessHint	✓	1	1
BFHint	✗	d	b^d
heuristicHint	✓	d	b^d

Table 1: Comparison of the hint algorithms

do this by taking several problems from the domains listed in Section 2 as a test set. We first transform these problems into problems in our DSL and then run the different implementations. From the results we gather in this process, we areable to draw conclusions about our approach and the different implementations.

Acknowledgements

This research is supported by the Dutch Technology Foundation STW, which is part of the Netherlands Organization for Scientific Research (NWO), and which is partly funded by the Ministry of Economic Affairs.

References

1. B. Heeren, J. Jeuring, and A. Gerdes. Specifying rewrite strategies for interactive exercises. *Mathematics in Computer Science*, 3(3):349–370, 2010.
2. Johan Jeuring, Alex Gerdes, and Bastiaan Heeren. A programming tutor for haskell. In *Central European Functional Programming School - 4th Summer School, CEFP 2011, Budapest, Hungary, June 14-24, 2011, Revised Selected Papers*, pages 1–45, 2011.
3. Johan Jeuring, Frans Grosfeld, Bastiaan Heeren, Michiel Hulsbergen, Richta IJntema, Vincent Jonker, Nicole Mastenbroek, Maarten van der Smagt, Frank Wijmans, Majanne Wolters, et al. Communicate!a serious game for communication skills. In *Design for Teaching and Learning in a Networked World*, pages 513–517. Springer, 2015.
4. Stephen Lavelle. Puzzlescript, 2016.
5. Chong-U Lim and D. Fox Harrell. An approach to general videogame evaluation and automatic generation using a description language. In *2014 IEEE Conference on Computational Intelligence and Games, CIG 2014, Dortmund, Germany, August 26-29, 2014*, pages 1–8, 2014.
6. Rinus Plasmeijer, Bas Lijnse, Steffen Michels, Peter Achten, and Pieter W. M. Koopman. Task-oriented programming in a pure functional language. In *Principles and Practice of Declarative Programming, PPDP'12, Leuven, Belgium - September 19 - 21, 2012*, pages 195–206, 2012.
7. Stuart J. Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach (3. internat. ed.)*. Pearson Education, 2010.
8. Kurt VanLehn. The behavior of tutoring systems. *International Journal of Artificial Intelligence in Education*, 16(3):227–265, 2006.