Automated Program Construction Based on Previous Experience

Yulia Korukhova[1][1], Nikolay Fastovets,

[1]*Computational Mathematics and Cybernetics Faculty, Lomonosov Moscow State University*,
Leninskie Gory, GSP-1, Moscow, 119991,Russian Federation

yulia@cs.msu.su, niknik1200@gmail.com

Abstract. We discuss the automated program construction using case-based reasoning approach. The derivation of a program starts from a logic-based specification. The case library summarise the previously obtained experience, it consists of a set of already known specifications and corresponding program codes. We are looking for a similar specification in the library and try to adapt the corresponding program code to get the solution. The main problems that occur in the implementation of the approach are the following: organization of the case library, definition of similarity and adaptation. We propose to keep the case library as an ontology, the ALC is used to describe specifications. This representation helps us to find similar specifications and to adapt the corresponding solutions as demonstrated on some examples.

Keywords: Automated program synthesis from specifications, case-based reasoning, description logics, ALC, ontologies.

1 Introduction

The problem called automation of programming was treated almost in the same time with the development of programming languages. First, the idea was to come to the higher level of abstraction: from programming in the machine codes and the direct use of assemblers to the higher level of abstraction – to programming languages. The development of a program often starts from the specification – from a description what the program should do. Writing specifications instead of programs looks like using the next level of abstraction in programming. Good specification is more clear and readable for the user. The development of automated synthesis systems aims to create such a language (or languages).

There are several approaches to automated construction of programs. Deductive methods [8] intend the construction of programs simultaneously with proof of their specifications. There are several synthesis methods that use this approach, but the automation of deductive synthesis is a challenging task: the success depends not only on the correctness of the given specification, but also on the sufficiency of domain information (usually described as a set of axioms or partially included in the synthesis rules) and on the order in which the derivation rules are applied. In practical applications some additional heuristics, and combinations of several methods are required to reduce the extremely large search space.

Another idea used in AI works is to decompose a difficult tasks into smaller (and simpler) ones and to compose the final solution from the derived (or even known) sub-tasks' solutions. In this case the synthesis task may be reduced to the task of planning and methods like Hierarchical Task Networks (HTN) [6] can be used. For example, if we need to construct an ordered list of elements that are presented in the two given sets, we may decompose the task into two parts: finding the intersection of two given sets and the task of construction of an ordered list from the set.

The idea of this article was inspired by observations how students who've started to learn programming solve some of the tasks. They have a course on basic algorithms, so they know several examples of programs. If they are asked to solve a new task they sometimes invent a new algorithm, but in many cases they either try to reduce the given task to a set of already discussed ones (as in the HTN approach) or they try to adapt an example program they have to the requested one. The precondition for application of the last method is the following: the students should have a solution for some task that is

[1] The work has been partially supported by RFBR grant 08-01-00627.

considered to be similar. In this work we are trying to model this process of solving tasks in programming. The similar approach has been proposed for software design patterns [11]. We are trying to implement case-based reasoning (CBR) for construction of programs from their descriptions based on first-order logic.

The paper is organized as follows: in the section 2 the foundational methods and their application for a particular task of program synthesis are discussed together with the problems that arise. In the section 3 our approach to the solution of the mentioned problems is presented, our prototype system and some results are described. In the section 4 the conclusions are drawn.

2 Basic Methods for Synthesis

In the task of synthesis we have a specification, that describes a relationship between inputs and outputs of a program (but it does not necessarily describes any algorithm for results' computation) and we need to construct a program on some programming language, that implements a computational algorithm for an output that meets all the conditions from the specification. We are going to explore a case-based reasoning approach.

2.1 Case-Based Reasoning

Humans often use some set of known solutions for different tasks and they try to adapt these solutions for new problems they are faced with.

The general CBR is performed as a cycle of the following four steps [1]:

1. RETRIEVE the most similar of the cases.
2. REUSE the information and knowledge in the found case to solve the new problem (by adapting the solution of the similar case).
3. REVISE the proposed solution.
4. RETAIN the parts of this experience likely to be useful for future problem solving.

The main feature of this approach is the ability of creation of a self-learning system that allows to improve performance of work with case library and improves adaptation methods. However, this approach has some problems. First, the case library is very large on practice, and if we have rather significant delays while searching a case in the library, such a system became useless. Secondly, the correctness of work of a system depends on definition of similarity between cases. Also adaptation of a solution is a challenging task. To use this approach for program synthesis these particular problems are needed to be solved.

2.2 Ontologies and Description Logics

In our system we propose to organize the case library as an ontology. This allows to use the mechanisms developed for ontologies for comparing specifications of functions and helps to find similar cases for the given tasks.

"An ontology is a logical theory accounting for the intended meaning of a formal vocabulary, i.e. its ontological commitment to a particular conceptualization of the world" [2]. We can define ontology as a systematic structure of knowledge, which describes the classification of some objects and relationships between objects. In our case the specifications of already known functions are such objects.

The most important for us is the classification described in the so-called terminological box (TBox). In many cases, descriptions of terms in the ontology used by so called description logic. Expressions (concepts) in such logics describe some subset of elements of the original set (the alphabet). We will consider one of such logics - ALC [3], which we use in our system.

2.3 Description Logic ALC

ALC is one of family of description logics. In this section we recall syntax and semantics of ALC expressions [3].

Concepts expressions in ALC are built up from the set of atomic concepts C and roles R according to the following recursive definition

$$C ::= A \mid \neg C \mid C \wedge C \mid C \vee D \mid \forall R.C \mid \exists R.C \mid \top \mid \bot$$

where $A \in C$ and $R \in R$.

An interpretation (model) *I* for ALC defined as a pair (Δ^I, $\cdot^I$), where Δ^I is non-empty set (alphabet) and $\cdot^I$ is a function mapped every atomic concept C to set $C^I \subseteq \Delta^I$, and every role R to a pair $R^I \subseteq \Delta^I \times \Delta^I$. The rules of interpretation for complex concepts are defined in the table 1.

Table 1: Interpretation of complex concepts in ALC.

Expression	Interpretation
$C \wedge D$	$C^I \cap D^I$
$C \vee D$	$C^I \cup D^I$
$\neg C$	$\Delta^I \setminus C^I$
$\exists$ R.C	$\{ x \in \Delta^I \mid \exists y\ R(x, y) \text{ and } y \in C \}$
$\forall$ R.C	$\{ x \in \Delta^I \mid \forall y\ R(x, y) \rightarrow y \in C \}$
T	Δ^I
$\bot$	Ø

A concept C is said to be satisfiable if there is a model *I* such that $C^I \neq$ Ø. If there are no such a model for the concept it is called unsatisfiable. A concept C is said subsumed by D (or D subsumes C) if for every model *I* $C^I \subseteq D^I$. A concepts C and D equivalent if C subsumes D and D subsumes C.

The ALC is used for internal description of specification in the system. The specifications presented in this form is easier to compare [10]. In our work we also order predicates in the ALC-specifications lexicographically.

2.4 Using of Ontologies in CBR

The ability to use ontologies for organizing the knowledge base became very useful for storing data and also in CBR. The system Taaable [5] can be considered as an example of such use. In this system an ontology is used to store information on a set of known recipes, as well as for selection of possible ways of adaptation of known solutions to new problems.

The hierarchy concept used in the ontology allows to reduce the number of possible precedents. A description of the concept itself provides an opportunity to advance the conclusion of the existing differences between the known precedents and the goal. Thus, this approach helps to organise our case library hierarchically that is useful both for the search and for the adaptation process.

We are going to use the similar idea for organizing the information about programs and their specifications.

3 Implementation and Current Results

In this section we observe the implementation of the proposed approach in our prototype synthesis system.

3.1 Common model of work

Pursuant to idea of the general CBR cycle described in section 2.1 we propose the following synthesis workflow:

1. Reading a specification for target program.

2. Search for the most similar specification among the already known ones.

3. Adaptation of a found program for the current task.

4. The derived program is presented to user for getting some feedback (presented for verification).

5. The specification and the derived program are added to the case library.

At the first stage our system loads the definitions for all known predicates and functions, loads programs and performs deserialization of concepts hierarchy. After that the system waits for a specification to work with.

When a specification for target program is given the system translates it to ALC-description and performs search for an exactly matching or the most similar specification of an already known program in the ontology using the search algorithm that will be decribed in the section 3.6. If the system finds a specificaion that is the exactly the same, the corresponding program is considered to be the solution and no adaptation is required. Otherwise the system looks for the most similar specification and then the system will start adaptaion of program associated with it.

At the adaptation step the system builds a description of the differences between the specification given by user and the similar specification found in the case library. Next, system perform search of similar known differences descriptions, using our search algorithm. If there are unknown differences the system reports about it to user and asks for a new adaptation rule. If the necessary adaptation rules are found, they are used to adapt the solution from the case library.

At the current stage of work the resulted program is given to user and he needs to verify it (but this step is also planned to be automated). If the solution is considered correct, the specification and the text of the program are added to the ontology – to the knowledge base of the system.

3.2 Knowledge Base Organization

Our system uses three libraries: for the programs, for the adaptation rules and for the definations of predicates and functions. The main part is the ontology which contains hierarchies of functions' specifications, differences descriptions, types, constants, special concepts (like *Output*, *Input*) and relations. Every specification in the ontology is connected with program code of the corresponding function (which is stored in the library of functions). Also, each description of differences is connected with one of adaptation rules.

The system takes as a task a specification based on the first order logic, because it is easier to be understood by users. In the internal form the specifications are stored as concepts written on ALC. The translation into this form is done by the system automatically.

Every function's name is associated with some predicate name (for removing functions during first-order logic to ALC translation process), and every predicate associated with some concept in the ontology. We use the terminology box (TBox) of ontology to perform search and to compare specifications.

Hierarchical organization in the ontology allows to reduce the search time, which is an important advantage for CBR approach in program synthesis problem solving.

In addition to this branches of hierarchy we use types hierarchy for to describe the relationship between the types of variables, roles hierarchy and concepts, associated with constants.

For work with this knowledge the following algorithms are used.

3.3 Building Concepts for Functions

After receiving a specification from user, the system should search for similar ones among the known functions. Thus, our first task is to build a sufficiently informative description from the specification function. To illustrate the translation method let's take a simple example of a square root specification: $\forall x.Real\ \exists y.Real\ (x>0)$ and $(y=sqr(x))$

1. After obtaining the expression S the specifying the goal function, we translate it into disjunctive normal form and make the whole expression to be the scope of the quantifiers

$$S' = [\text{Quantifiers}\ ]\ D_1 \vee \ldots \vee D_n,$$

For our example of square root no transformation is needed at this stage.

2. We replace all the function calls for each clause D_i in the obtained expression S' by new variables (that don't occur free in the expression before this stage) and add predicates that correspond to these functions. For our example of square root we obtain the following expression:

$\forall x.\text{Real}\ \exists y.\text{Real}\ (x>0) \wedge (y=v1) \wedge \text{Sqr}(v1,x)$

where Sqr(v1,x) is true if and only if v1=sqr(x), this predicate corresponds to sqr function.

3. After the previous step we have a set of clauses, each of them represents an atom or a conjunction of atoms, where the arguments are variables. Now we can go directly to the construction of the ALC concept for the function. In such description we will use special roles: *Contain*, *ContainNeg*, *Variant* and the family of roles $Param_1$, …, $Param_n$. Also, we will use special concepts, which describe the roles of variables in the specification: *Output*, $Input_1$, …, $Input_m$, $Variable_1$, …, $Variable_p$.

First of all we associate every variable in S' with concept of form (*VarRole* $\wedge$ *VarType*), where VarRole is one of special concepts Output, Input1 etc. and VarType is concept of domain for variable. For example, concept for integer output variable will have form (*Output* $\wedge$ *Integer*). Note that different variables will be associated with different concepts.

Then, for each atom $A(x_1, \ldots, x_k)$ we build a description

$$AC = cA \wedge \exists Param_1.x_1 \wedge \ldots \wedge \exists Param_k.x_k,$$

where cA is concept associated with predicate A and variables x_1, …, x_k replaced with variable descriptions (as above).

Next step is building of concept for conjunction. We describe conjunction as set of intersections between concepts P_1, …, P_L, each of them has form ∃Contain.AC, if corresponding atom contain in expression with positive polarity, or ∃ContainNeg.AC – if it has negative polarity. So, we get conjunction description

$$CC = \exists Contain.AC \wedge \ldots \wedge \exists ContainNeg.AC' \wedge \ldots$$

For the same example of the square root specification we obtain the following concept:

$\text{Func} \wedge \exists\text{Contain}.(> \wedge \exists\text{Param1}.(\text{Input1} \wedge \text{Real}) \wedge \exists\text{Param2}.(\text{Zero})) \wedge \exists\text{Contain}.(= \wedge \exists\text{Param1}.(\text{Output} \wedge \text{Real}) \wedge \exists\text{Param2}.(\text{Var1} \wedge \text{Real})) \wedge \exists\text{Contain}.(\text{Sqr} \wedge \exists\text{Param1}.(\text{Var1} \wedge \text{Real}) \wedge \exists\text{Param2}.(\text{Input} \wedge \text{Real}))$

If the expression contains a disjunction then we build descriptions for each disjunct separately and include them in the common description: $\exists Variant.CC_1 \wedge \exists Variant.CC_n$. Thus we obtain a complete description of the specification S. The specification in this form is compared with other ones from the case library.

3.4 Difference Concept Building

To store and explore our knowledge about adaptation for programs we use another kind of concepts - the differences concepts, which are also written on ALC and organized as an hierarchy. Every such concept describes a subset of adaptation rules which eliminates the corresponding differences in specifications and transforms the program text from the case library.

We build such differences descriptions as concepts with special roles: Add and Remove or their intersection. Our difference concepts building algorithm consist of the following steps.

1. The input data for procedure are the two concepts of functions A and B.

2. We split A and B into two sets A' and B', which consists of the subdescriptions from A and B relatively. So, every item of this sets is concept in form ∃Contain.AC or ∃ContainNeg.AC, where AC is description of atom. Such items we mark as A'_i and B'_i relatively.

3. Next, we build concepts $\exists Add.B'_i$ for each item of B which not contained in A' and $\exists Remove.A'_i$ for each item of A which not contained in B'. Intersection of such concepts is the newly build concept of differences that is used in the solution adaptation process.

3.5 Adaptation

The adaptation is needed if we do not have the case exactly matching the given problem. For the similar (but not the same) case we usually need to build a concept of differences and to adapt the solution. Our adaptation mechanism uses hierarchy of differences concepts and a list of adaptation rules, each of them is associated with one of differences concept.

Our adaptation mechanism based on applying of rewriting rules to known program. A part of our ontology contain descriptions of differences between specifications. Such concepts descripts rewriting of specifications. Them constructed with special roles: Add, Remove, Replace, From, To, InVariant.

For example, consider simple clauses:
$\exists$Contain.(Rel1 $\wedge$ $\exists$Param1.(Temp $\wedge$ Var1 $\wedge$ Integer))
and
$\exists$Contain.(Rel1 $\wedge$ $\exists$Param1.(Const_1 $\wedge$ Integer))
Differences will be descripted as follow:
$\exists$Replace.($\exists$From.(Temp $\wedge$ Var1) $\wedge$ $\exists$To.(Const_1))
The presented adaptation says that the replacement of a variable by a constant of the same type should be dome for this example.

Thus, we can build such descriptions of differences and use them for search for required rewriting rules. Rewriting rules can be strictly described or they can have form of patterns.

So, adaptation algorithm consists of the following steps.

1. Input data for the procedure are: known function concept A, a program P associated with it and new specification S.

2. First of all we build concept C for specification of new function S.

3. Search algorithm returns exactly matching or a most similar to it concept from the case library. We mark it as K.

4. If we do not have the exact matching we build differences concept D for C and K.

5. At the next step we search for D in the hierarchy of differences concepts. If it is not found, we split D into subconcepts by intersection and make search for this subconcepts.

6. If the initial difference concept D or all its subconcepts are found, we apply the adaptation rules (rewriting rules) associated with them to the program P. The program P', obtained after rewriting, is the proposed solution. If the appropriate difference concepts are not found, the system returns negative answer and requests for a new adaptation rule.

3.6 Concepts comparison

Our comparison algorithm based on structural subsumption [4]. We use next comparing rule: for two concepts
$A = A_1 \wedge \ldots \wedge A_n$ and $B = B_1 \wedge \ldots \wedge B_m$,

if $\forall i$, $1 \leq i \leq n$ ($\exists j$, $1 \leq j \leq m$ such that $A_i \sqsubseteq B_i$)

then $A \sqsubseteq B$.

So, our subsumption checking algorithm based on this rule return answer $A \sqsubseteq B$ if and only if for each subconcept from A there is a subconcept in B that subsumes it.

Respectively to this assessment we build our functions and differences concept hierarchy. So, we can postulate that every concept C that subsumes by some concept D contain all restrictions from D. Also, we define similarity between concepts C and D as the length of path in the hierarchy tree from C to D. According to this the most similar concept for a given one (functions or differences) is it's immediate predecessor in our library.

3.7 Search Algorithm

Our search algorithm uses the defined above subsumption assessment as an heuristic in breadth-first search. The target concept and the root of hierarchy tree (or another node, which can be used as root of subtree) are inputs in the process. In search process we generalize the target clause by replacing of concepts associated with the constants of the overall concept that brings together all known system constants. Note, that type's concepts are not changed. Thus, we eliminate probable mistakes deals with differents in constants only (if differences between specifications are in constants only then it is easy to obtain new program by replacing constants).

At the every step we check matching between the root concept and the target. If our search procedure finds exact matching, it returns the root. In the other case, it checks the subsumption assessment between target and all the direct descendants of the root. If one of the descendants subsumes the target, our search procedure performs recursive call. If there are no target subsumers between root descedants, procedure returns root as the answer (as the most similar concept).

Since we check occurrences for all subconcepts (that means all the restrictions) in our subsumption assessment, we can guarantee that the only one of roots descendants is the predecessor for our target.

3.8 An Example of Synthesis

Let's consider an example of synthesis. The system was asked to build a program that immitate the functioning of a ticket machine. The machine takes 1, 2 and 3 coins and gives the ticket when the inserted amount of money is 3. If it is greater than 3, the machine returns the money back. If the entered amount is smaller than 3 the machine just keeps waiting. The task can be formalized as following: in the program we need to model a functioning of a finite state automata with 4 states: T (the ticket is given), Noth1(doing nothing, the money 1 was already inserted), Noth2 (doing nothing, the amount of money inserted equals 2), MR (money return) and 12 transitions. In the specification we are going to use the predicate InState(a,b) which is true if and only if the automata is in the state a then the value of input variable is b.The corresponding specification is the following:

$$
\begin{aligned}
\forall m\, \exists y :\ & \mathrm{InState}(MR, m) \wedge \mathrm{InState}(y,m+3) \wedge y = T \vee \\
& \mathrm{InState}(MR,m) \wedge \mathrm{InState}(y,m+1) \wedge y = Noth1 \vee \\
& \mathrm{InState}(MR,m) \wedge \mathrm{InState}(y,m+2) \wedge y = Noth2 \vee \\
& \mathrm{InState}(Noth1,m) \wedge \mathrm{InState}(y,m+1) \wedge y = Noth2 \vee \\
& \mathrm{InState}(Noth1,m) \wedge \mathrm{InState}(y,m+2) \wedge y = T \vee \\
& \mathrm{InState}(Noth1,m) \wedge \mathrm{InState}(y,m+3) \wedge y = MR \vee \\
& \mathrm{InState}(Noth2,m) \wedge \mathrm{InState}(y,m+1) \wedge y = T \vee \\
& \mathrm{InState}(Noth2,m) \wedge \mathrm{InState}(y,m+2) \wedge y = MR \vee \\
& \mathrm{InState}(Noth2,m) \wedge \mathrm{InState}(y,m+3) \wedge y = MR \vee \\
& \mathrm{InState}(T,m) \wedge \mathrm{InState}(y,m+1) \wedge y = Noth1 \vee \\
& \mathrm{InState}(T,m) \wedge \mathrm{InState}(y,m+2) \wedge y = Noth2 \vee \\
& \mathrm{InState}(T,m) \wedge \mathrm{InState}(y,m+3) \wedge y = T
\end{aligned}
$$

Each conjunction in the expression above corresponds to one transition of the automata. The system translates the specificaion into ALC description and starts the search in the case library. There is no specification that is exactly the same as the given one. Suppose, that our ontology already contains the specifications and the corresponding programs for a turn-signal controller, for a garland controller and for a traffic light controller.

The turn-signal controller turns the signal off after 2 seconds of working and then in one second it turns the signal on again. It has the following specification (it's translation to ALC and the corresponding program are stored in the case library):

$$
\begin{aligned}
&\forall t\, \exists y : \mathrm{InState}(Const\_On, t) \wedge \mathrm{InState}(y, t+2) \wedge y = Const\_Off \vee \\
&\mathrm{InState}(Const\_Off, t) \wedge \mathrm{InState}(y, t+1) \wedge y = Const\_On
\end{aligned}
$$

The garland controller changes the colours of the garland lamps from blue to orange then from orange to purple and then back to the blue colour. Each colour stays on for some time from 1 to 5 seconds (the time is chosen randomly). Such controller has the following specification:

$\forall t\ \exists y$: InState(Const_Blue, t) $\wedge$ InState(y, t+random(1, 5)) $\wedge$ y = Const_Orange $\vee$
InState(Const_Orange, t) $\wedge$ InState(y, t+random(1, 5)) $\wedge$ y = Const_Purple $\vee$
InState(Const_Purple, t) $\wedge$ InState(y, t+random(1, 5)) $\wedge$ y = Const_Blue;

The traffic light can be represented as a finite state automata with 3 states (Red,Yellow and Green) and 3 transitions. We assume that red light is switched on for 10 seconds, then the yellow is on for 2 seconds, after that the green light is on for 10 seconds, then again the light turns to red. The specification for a traffic light controller is the following:

$\forall t\ \exists y$: InState(Red, t) $\wedge$ InState(y,t+10) $\wedge$ y =Yellow $\vee$

InState(Yellow,t) $\wedge$ InState(y,t+2) $\wedge$ y= Green $\vee$

InState(Green,t) $\wedge$ InState(y,t+10) $\wedge$ y= Red

Hierachy of this part of the ontology has the form:

<turn-signal controller> ⊆ <garland controller>

<turn-signal controller> ⊆ <traffic-light controller>

Thus, turn-signal controller is the most general case for this part of ontology. Suppose, that we come to this node on the search stage. First of all we check subsumption relation between traffic-signal controller and our target description. The system notice, that all the subconcepts (restrictions from the specification) can be found in the target clause. Thus, turn-signal controller program can be a candidate for a similar specification. However, we have to check the next nodes.

The garland controller contains similar subconcepts, but also it has a new subconcept, assotiated with the function *random*, while the target clause has neither restrictions like it nor a variable, that can be replaced by this function. As for the given traffic-light description all it's subconcepts are presented (or can be mapped) in target description. Thus, traffic-light is considered to be more similar to our target than the turn-signal. The traffic light specification doesn't have any descendants in the ontology, hence the system use the program for a traffic light as a base for construction of the target program.

To obtain the solution we need to perform an appropriate adaptation. The differences between cases are the number and names of states and the values of m and t which transition depends on. We are trying to find and remove differences in the target specificaion and in the most similar one from the case library. We have to do the following replacements:
in the first disjunct Red → MR; 10 → 3; Yellow → T;
in the second disjunct: Yellow →T; t→m; 2 → 1; Green → Noth1;
in the third disjunct: Green → Noth1; t → m; 10 → 3;
Also, we need to add all the disjuncts, that can not be obtained with such replacements only.

Such replacements are described as a differences concept. Our next step is search for the same differences concept or for a set of it's subconceptst. The replacements of constants can be performed with one pattern [Const_X → Const_Y], because the constants have the same type. The differences in variables are related to function calls. We can check, that all the different variables obtained by eliminating of function calls are connected with the same functions. Thus, we can leave them unchanged (we will not meet such variables in the program). At the same time we can not be certain that we can find exactly matching differences concept that describe adding of all conditional branchings for new machine statements. Hence, we can use pattern, which assotiated with such subconcept.

Suppose, that traffic-light program has the form:
```
void TrafficLight ( void ) {
    int State = Red;
    int dt;
    while(1)    {
       cin >> t;
      switch(State){
       case Red:  if(t == 3)State = Yellow; break;
       case Yellow: if(t == 1)State = Green; break;
       case Green: if(t == 10)State = Red; break;
      }
    }
}
```

```
Replacement of constants will give us:
/*...*/
switch(State){
case MR:
    if(t == 3)State = T;
    break;
case T:
    if(t == 2)State = Noth1;
    break;
case Noth1:
    if(t == 3)State = MR;
    break;
}
/*...*/
```

Next step is adding of the new conditional branches with pattern
[case Constant1: if(Var1 == Constant2) Var2 = Constant3; break;]
associated with the following concept:

∃Add. (∃ Contain.(InState /\ ∃Param1.(Constant1) /\ ∃Param2.(Var1)) /\ ∃ Contain.(InState /\ ∃Param1.(Var2) /\ ∃Param2.(Func /\ Var8)) /\ ∃ Contain.(Sum /\ ∃Param1.(Func /\ Var8) /\ ∃Param2.(Temp /\ Var1 /\ Int) /\ ∃Param3.(Constant2)) /\ ∃ Contain.(Equal /\ ∃Param1.(Int /\ Var2 /\ Temp) /\ ∃Param2.(Int /\ Constant3)));

We obtain:

```
/*...*/
switch(State){
case MR:    if(t == 3)State = T; break;
case T:     if(t == 2)State = Noth1; break;
case Noth1: if(t == 3)State = MR; break;
case Noth2: if(t == 3)State = MR;
    break;
/*...*/
}
/*...*/
```

Thus, we obtain a new program for a trade machine from the known program for a traffic-light.

4 Conclusions

In this paper we consider one of possible applications of Case-Based Reasoning - for the program synthesis task. Using of ontologies here allows us to organize search in the case library in a reasonable time and helps to define cases' similarity. The same idea applied to differences concepts helps to formalize the search of differences and perform adaptation. The ideas were tested on several examples, but the work is still in progress. First, we are going to extend the case library. Another point of improvement is the way of verification of the obtained program. Now this question is solved by user, but it is planned to be also automated. Also for construction of a program two or more existing solutions may be useful, we are going to try to exploite technique like in [9]. We are still far away from fully automated synthesis, but this approach makes a step to this goal.

5 References

1. A. Aamodt, E. Plaza Case-Based Reasoning: Foundational Issues, Methodological Variations, and System Approaches. AI Communications. IOS Press, Vol. 7: 1 (1994)

2. N. Guarino Formal Ontology and Information Systems. Amsterdam, IOS Press, pp. 3-15 (1998)

3. M. Bienvenu Prime Implicate Normal Form for ALC Concepts. Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (2008)

4. F. Baader, W. Nutt Basics Description Logics. The Description Logic Handbook: Theory, Implementation and Application (2003).

5. F. Badra, J. Cojan, A. Cordier, J. Lieber, T. Meilender, A. Mille, P. Molli, E. Nauer, A. Napoli, H. Skaf-Molli, Y. Toussaint Knowledge Acquisition and Discovery for the Textual Case-Based Cooking system WIKITAAABLE. 8th International Conference on Case-Based Reasoning - ICCBR 2009, Workshop Proceedings, Seattle : United States (2009)

6. K. Erol, J. Hendler, and D. Nau HTN Planning: Complexity and Expressivity. In Proceedings of AAAI-94 (1994)

7. Y. Korukhova An approach to Automatic Deductive Synthesis of Functional Programs. Annals of Mathematics and Artificial Intelligence, Vol. 50, Numbers 3-4 – Springer Netherlands (2007)

8. Z. Manna Z. and R. Waldinger Fundamentals of Deductive Program Synthesis. IEEE Transactions on Software Engineering, 18(8), (1992)

9. S. Ontañón and E. Plaza Amalgams: a Formal Approach for combining Multiple Case Solutions. //Case-Based Reasoning. Research and Development, LNCS, Vol. 6176, 2010

10. S. Ontañón and E. Plaza On Similarity Measures Based on a Refinement Lattice// Case-Based Reasoning Research and Development, LNCS Vol. 5650, 2009

11. P. Gomes, F. C. Pereira, P. Paiva, N. Seco, P. Carreiro, J. L. Ferreira, C. Bento Using CBR for Automation of Software Design Patterns //Advances in Case-Based Reasoning , LNCS Vol. 2416, 2002