2011 Eighteenth International Symposium on Temporal Representation and Reasoning

Algorithm Design Template base on Temporal ADT

Nikolay Shilov
A.P. Ershov Institute of Informatics Systems
Russian Academy of Sciences
Novosibirsk, Russia
Email: shilov@iis.nsk.su

***Abstract*—Design and Analysis of Computer Algorithms is a must of Computer Curricula. It covers many topics that group around several core themes. These themes range from data structures to complexity theory, but one very special theme is algorithmic design patterns, including greedy method, divide-and-conquer, dynamic programming, backtracking and branch-and-bound. Naturally, at the undergraduate level all listed design patterns are taught, learnt and comprehended by examples. But they can be semi-formalized as design templates, semi-specified by correctness conditions, and semi-formally verified, for example, by means of Manna – Pnueli proof-principles. Moreover, this approach can lead to new insights and better comprehension of the design patterns, specification and verification methods. In this paper we demonstrate an utility of the approach by study of backtracking and branch-and-bound design patterns. In particular, we present, specify and prove correctness of the unified template for these patterns. Our approach is based on a temporal abstract data type *theque* that unifies stack and queue discipline. We prove that every algorithm instantiated from the template is totally correct, if the input graph for traversing is finite, the boundary condition is monotone, and the decision condition is anti-monotone on sets of "visited" vertices.**

***Keywords*-backtracking; branch-and-bound; temporal abstract data type; safety properties; liveness properties; temporal proof principles; total correctness;**

I. Introduction

Graph traversal refers to the problem of visiting all the nodes in a (di)graph to find particular nodes (vertices) that enjoy some property specified by some Boolean *criterion condition* C. A Depth-first search (DFS) is a technique for traversing a finite graph that visits the child nodes before visiting the sibling nodes. A Breadth-first search (BFS) is another technique for traversing a finite undirected graph that visits the sibling nodes before visiting the child nodes.

Sometimes it is not necessary to traverse all vertices of a graph to collect the set of nodes that meet the criterion function, since there exists some Boolean *boundary condition* B which guarantees that child nodes do not meet the criterion function C: Backtracking (BTR) is DFS that uses boundary condition, branch-and-bound (B&B) is DFS that uses boundary condition. Backtracking became popular in 1965 due to research of S.W. Golomb and L.D. Baumert [5], but it had been suggested earlier by D. H. Lehmer. Branch-and-bound was suggested in 1960 by A.H. Land and A.G. Doig [6].

Unfortunately, further progress with BTR and B&B techniques has degenerated into an extensive collection of "success stories" and "recipes" how they have been used in the context of particular combinatorial or optimization problems. This leads to educational situation when the most popular contemporary textbooks on the algorithm design and implementation look like Cooking Books [1], [3]. The situation has become similar to a situation with loops in programming languages: *while*-, *until*-, and *for*-loop patterns are known since early 1960s, but more problem-oriented loop templates can improve program efficiency and simplify code verification [9].

The rest of the paper is organized as follows. A special temporal abstract data type that unifies stack and queue discipline is described in the next section II. The unified template for backtracking and branch-and-bound design patterns is presented in the section III altogether with a backtracking algorithm for the classical folklore n-Queen Puzzle as a very simple example of specialization of the unified template. Section IV presents an example of a problem-oriented template instantiated from the unified template; this instantiated template implements branch-and-bound design pattern for the discrete knapsack problem. Specification of the unified template (in Floyd – Hoare style) along with a proof-sketch of the total correctness (with aid of Manna – Pnueli temporal proof-principles) are given in the section V. We conclude by discussion of some directions for further research in the last section VI.

II. Temporal ADT Theque

Let us define a special temporal abstract data type (ADT) *theque*[1] for the unified representation of BTR and B&B. Theque is a finite collection (i.e. a set) of values (of some background data type) marked by disjoint *time-stamps*. The time stamps are readings of *global clock* that counts time in numbers of *ticks*, they (time-stamps) never change and always are not greater than current reading of the clock. Let us represent an element x with a time-stamp t by the pair (x, t). Readings of the clock as well as time-stamps are not visible for any observer. Let us assume that this tick is

[1] *Theque* from Greek *theke* – storage, repository, e. g. *discotheque*.

1530-1311/11 $26.00 © 2011 IEEE
DOI 10.1109/TIME.2011.18

indivisible, every action takes a positive (integer) number of ticks, and the clocks never resets or restarts.

ADT theque inherits some set-theoretic operations: the *empteq* (i.e. *empty theque*)) is simply the empty set ($\emptyset$), set-theoretic equality ($=$) and inequality ($\neq$), subsumption ($\subset$, $\subseteq$). At the same time ADT theque has its own specific operations, some of these operations are time-independent, some others — time-sensitive, and some are time-dependent. Let us enumerate below time-independent operations, and describe time-dependent and time-sensitive operations in the next paragraphs.

- Operation Set: for every theque T let $Set(T)$ be $\{x : \exists t((x,t) \in T)\}$ the set of all values that belongs to T (with any time-stamp).
- Operations In and Ni: for every theque T and any value x of the background type let $In(x,T)$ denote $x \in Set(T)$, and let $Ni(x,T)$ denote $x \notin Set(T)$.
- Operation $Spec$ (*specification*): for every theque T and any predicate $\lambda x.Q(x)$ of values of the background type let theque $Spec(T,Q)$ be the following sub-theque $\{(x,t) \in T : Q(x)\}$.

The unique time-dependent operation is a synchronous addition $AddTo$ of elements to theques. For every finite list of theques T_1, ... T_n ($n \geq 1$) and finite set $\{x_1, \ldots x_m\}$ of elements of the background type ($m \geq 0$), let execution of $AddTo(\{x_1, \ldots x_m\}, T_1, \ldots T_n)$ at time t (i.e. the current reading of the clock is t) returns n theques T'_1, ... T'_n such, that there exist m moments of time (i.e. readings of the clock) $t = t_1 < \ldots < t_m = t'$ such that t' is the moment of termination of the operation, and for every $1 \leq i \leq n$ the theque T'_i expands T_i by $\{(x_1,t_1), \ldots (x_m,t_m)\}$, i.e. $T'_i = T_i \cup \{(x_1,t_1), \ldots (x_m,t_m)\}$. Let us observe that this operation is non-deterministic due to several reasons: first, the set of added elements $\{x_1, \ldots x_m\}$ can be sorted in different manners; next, time-stamps $t_1 < \ldots < t_m$ can be arbitrary (starting at the current time). Let us write $AddTo(x, T_1, \ldots T_n)$ instead of $AddTo(\{x\}, T_1, \ldots T_n)$ in the case of a singleton set $\{x\}$.

There are three pairs of time-sensitive operations: Fir and $ReMFir$, Las and $RemLas$, Elm and $RemElm$. Let T be a theque. Recall that all values in this theque have disjoint time-stamps.

- Let $Fir(T)$ be the value of the background type (i.e. without a time-stamp) that has the smallest (i.e. the first) time-stamp in T, and let $RemFir(T)$ be the theque that results from T after removal of this element (with the smallest time-stamp).
- Let $Las(T)$ be the value of the background type (i.e. without a time-stamp) that has the largest (i.e. the last) time-stamp in T, and let $RemLas(T)$ be the theque that results from T after removal of this element (with the largest time-stamp).

We also assume that $Elm(T)$ is *some* element of T (also without any time-stamp) that is defined according to some procedure (undisclosed for us) and $RemElm(T)$ is the theque that results from T after removal of this element (with its time-stamp).

III. Unified Template

Let us introduce some notation that unifies representation of BTR and B&B by a single template for graph traversing: let FEL and REM stay either for Fir and $ReMFir$, or for Las and $RemLas$, or for Elm and $RemElm$. It means, for example, that if we instantiate Fir for FEL, then we must instantiate Fir for FEL and $RemFir$ for REM throughout the template. Instantiation of Fir and $RemFir$ imposes a queue discipline *first-in, first-out* and specializes the unified template to B&B template; instantiation of Las and $RemLas$ imposes a stack discipline *first-in, last-out* and specializes the template to BTR template; instantiation of Elm and $RemElm$ specializes the unified template to "Deep Backtracking" or "Branch and Bounds with priorities" templates.

Let us say that a (di)graph is concrete, if it is given by the enumeration of all vertices and edges, or by the adjacency matrix, or in any other explicit manner. In contrast, let us say that a (di)graph G is virtual, if the following features are given:

- a type $Node$ of vertices of G, and the initial vertex ini of this type $Node$ such that every vertex of $u \in G$ is reachable from ini;
- a computable function $Neighb : Node \to 2^{Node}$ such that for any vertex of $u \in G$ it returns the set of all its neighbors $G(u)$ (children in a (di)graph).

In this notation an unified template for traversing a virtual graph G with aid of "easy to cheque"

- a *boundary* condition

$$B : 2^{Node} \times Node \to BOOLEAN,$$

- and a *decision* condition

$$D : 2^{Node} \times Node \to BOOLEAN$$

for collecting all nodes that meet a "hard to cheque"

- *criterion* condition $C : Node \to BOOLEAN$

can be represented by the following pseudo-code.

VAR U: Node;
VAR S: set of Node;
VAR Visit, Live, Out: theque of Node;
Live, Visit:= AddTo(ini, empteq, empteq);
Out:= empteq;
IF D({ini}, ini) THEN Out:= AddTo(ini, Out);
WHILE Live ≠ empteq
DO U:= FEL(Live); Live:= REM(Live);
S:= {W ∈ Neighb(U) :
Ni(W, Visit) & ¬B(Set(Visit), W)};
Live, Visit:= AddTo(S, Live, Visit);

Out:= Spec(Out, λx.D(Set(Visit), x));
IF D(Set(Visit), U) THEN Out:= AddTo(U,Out);
OD

Let us consider below a backtracking algorithm for the classical folklore n-Queen Puzzle (n-QP) as a simple example of specialization of the above unified template.

Let $n \geq 1$ be a given integer. The problem is to generate all "safe" placements of n queens on a generalized $n \times n$ chessboard, i.e. placements where no two queens *attack* each other (that is they do not share the same row, column, or diagonal).

Let us adopt as the virtual graph G the following tree of all *partial placements* (p-placements). Each p-placement is a safe placement of k ($0 \leq k \leq n$) queens on the first k rows. Let the empty placement *em-placement* (i.e. the placement of 0 queens on 0 first rows) be the root of the tree ini. Let function $Neighb$ compute for any p-placement of k queens on the first k rows ($0 \leq k < n$) all possible *extensions* by placing a new queen in any of n positions (columns) on the next row (that has number $(k+1)$).

The boundary, the decision, and the criterion conditions B, D and C are quite obvious: for every p-placement x of k queens on the first k rows ($0 \leq k \leq n$) let

- $B(x)$ be "x is not safe";
- $D(x)$ be "x is complete" (i.e. $k = n$);
- $C(x)$ be "x is compete and safe".

The backtracking algorithm for n-QP follows.

VAR U: p-placement;
VAR S: set of p-placements;
VAR Live, Out: theque of p-placements;
Live:= AddTo(em-placement, empteq);
Out:= empteq;
WHILE Live ≠ empteq
DO U:= Las(Live); Live:= RemLas(Live);
S:= {W extends U : W is safe};
Live:= AddTo(S, Live);
IF U is complete THEN Out:= AddTo(U,Out);
OD

IV. Example: Discrete Knapsack Problem

Below we present an example of specialization of the unified template to B&B template for the classical Discrete Knapsack Problem (DKP) [3]. More examples will follow in a forthcoming full publication of the research.

A. DKP problem statement

Let an integer $n \geq 0$ be the number of indivisible goods, non-negative real numbers $p_1 \geq 0$, ... $p_n \geq 0$ and $w_1 \geq 0$, ... $w_n \geq 0$ be their prices and weights, and a non-negative real number $W \geq 0$ be the maximal weight that is safe for a "knapsack". An admissible collection is a set of goods with the gross weight not more than W. The problem is to compute all admissible collections of goods each of which has the maximal total price among admissible collections.

Let us identify collections of goods with their characteristic vectors, i.e. vectors $(c_1, \dots c_n) \in \{0,1\}^n$. Let L and P be Load and Price functions defined as follows: for any $c_1, ...c_n \in \{0,1\}$ let $L(c_1, \dots c_n) = \sum_{1 \leq k \leq n} c_k \times w_k$ and $P(c_1, \dots c_n) = \sum_{1 \leq k \leq n} c_k \times p_k$. Then DKP can be formalized as follows: compute all collections $(c_1, \dots c_n) = \arg\max\{P(c_1, \dots c_n) : L(c_1, \dots c_n) \leq W\}$.

B. Branch and Bound for DKP

Let us adopt the complete binary tree T_n of all *partial collections* (p-collections) $(c_1, \dots c_m) \in \{0,1\}^m$, where $0 \leq m \leq n$, as a graph G for traversing, and the empty p-collection (the root of the tree) as the initial node ini. Function $Neighb$ is defined in a natural way: for every p-collection $c = (c_1, \dots c_m)$, $(0 \leq m < n)$ $Neighb$ computes two p-collections $c \circ 0 = (c_1, \dots c_m, 0)$ and $c \circ 1 = (c_1, \dots c_m, 1)$. Let us assume also that this tree is ordered by the lexicographical linear order $\preceq$ on p-collections.

Let us introduce further auxiliary notation. The length of a p-collection $(c_1, \dots\ c_m)$ is $|c_1, \dots c_m| = m$; the empty p-collection has length 0; let us remark that a collection is a p-collection of length n. Functions L and P can be naturally extended on p-collections: for any p-collection $(c_1, \dots c_m)$, $0 \leq m \leq n$, let $L(c_1, \dots c_m) = \sum_{1 \leq k \leq m} c_k \times w_k$ and $P(c_1, \dots c_m) = \sum_{1 \leq k \leq m} c_k \times p_k$. For every set of p-collections Q let $ext(Q)$ be the set of all *admissible extensions* of vectors in Q, i.e. the set $\{z \in T_n : L(z) \leq W$ and $x \preceq z$ for som $x \in Q\}$; let us write $ext(x)$ in the case when Q is a singleton $\{x\}$.

Let us define the boundary condition B as a function of two arguments: the first argument is a set X of p-collections, the second — some p-collection y; for any appropriate X and y let $B(X, y)$ be

a lower bound for $\max\{P(x) : x \in ext(X)\}$ $>$
$>$ an upper bound for $\max\{P(x) : x \in ext(y)\}$,

where

- a lower bound is computed according to some fixed method that is monotone on X (i.e. can not decrease while X is expanding as a set);
- an upper bound is computed according to some fixed method that is anti-monotone on y (i.e. can not increase while y is extending lexicographically).

For example, one can adopt

- $\max\{P(x) : x \in X$ and $L(x) \leq W\}$ as a method for computing a lower bound for $\max\{P(x) : x \in ext(X)\}$,
- $(P(y) + \sum_{|y|<k\leq n} p_k)$ as a method for computing an upper bound for $\max\{P(x) : x \in ext(y)\}$.

Other (more efficient) methods [3] are admissible also.

Let us define the decision condition D as a function of two arguments also: the first argument is a set X of p-collections,

the second — some p-collection y; for any appropriate X and y let $D(X, y)$ be a conjunction of the following two conditions:

- $P(y) = \max\{P(x) : x \in X \text{ and } L(x) \leq W\}$;
- y (i.e. $|y| = n$) is a collection and $L(y) \leq W$.

The criterion condition $C(x)$ is straightforward: x is a collection (i.e. $|x| = n$), $L(x) \leq W$ and $P(x) = \max\{P(y) : |y| = n \text{ and } L(y) \leq W\}$.

The unified template for branch and bound algorithm for DKP follows:.

VAR U: p-collection;
VAR S : set of p-collections;
VAR Visit, Live, Out : theque of p-collections;
Live, Visit:= AddTo(empty p-collection, empteq, empteq);
Out:= empteq;
WHILE Live $\neq$ *empteq*
DO U:= Fir(Live); Live:= RemFir(Live);
S:= $\{U \circ b : |U| < n$, *b*$\in \{0, 1\}$ *and*
lower bound for $\max$ $\{P(x) : x \in ext(Set(Visit))\}$
$\leq$ *upper bound for* $\max$ $\{P(x) : x \in ext(U \circ b)\}$;
Live, Visit:= AddTo(S, Live, Visit);
IF ($|U| = n$, *L(U)*$\leq W$ *and*
P(U)$> \max$ $\{P(x) : x \in Set(Out)\}$)
THEN Out:= AddTo(U, empteq);
IF ($|U| = n$, *L(U)*$\leq W$ *and*
P(U)$= \max$ $\{P(x) : x \in Set(Out)\}$)
THEN Out:= AddTo(U,Out);
OD

V. Specification and Correctness

An algorithm without specification is a tool without manual: no idea how to use it and what to expect. A specified algorithm without correctness proof is a non-certified tool, it can be dangerous in use. So we have to specify and prove correctness of our unified template. We would like to use Floyd – Hoare approach for specification [2], [4] and Manna – Pnueli temporal proof principles for proving [7], [8]. In Floyd – Hoare approach an algorithm is specified by a precondition and a postcondition for input and output data. In Manna – Pnueli approach *safety* properties are proved by induction, while *liveness* properties are proved with the aid of mappings to well-founded sets.

The *postcondition* is simple: Theque Out consists of all nodes of the graph G (with time-stamps) that meet the criterion condition C, and each of these nodes has a single entry (occurrence) in Out.

The *precondition* is more complicated and can be presented as a conjunction of the following clauses.

1) G is a virtual (di)graph, ini is a node of G, $Neighb$ is a function that computes for every node the set of all its neighbors so, that all nodes of G can be reached from ini by iterating $Neighb$.
2) For every node x of G the boundary condition $\lambda S.\ B(S, x)$ is a monotone function: $B(S_1, x)$ implies $B(S_2, x)$ for all sets of nodes $S_1 \subseteq S_2$ (i.e. if a node is ruled-out by a set, then it is ruled-out by any bigger set).
3) For all nodes x and y of G, for any set of nodes S, if y is reachable from x, then $B(S, x)$ implies $B(S, y)$ (i.e. if a node is ruled-out then all its successors are ruled out also).
4) For every node x of G the decision condition $\lambda S.\ D(S, x)$ is an anti-monotone function: $D(S_2, x)$ implies $D(S_1, x)$ for all sets of nodes $S_1 \subseteq S_2$ (i.e. a candidate node may be discarded later).
5) For every set of nodes S, if $S \cup \{x \in G : B(S, x)\}$ is equal to the set of all nodes of G, then $D(S, x) \Leftrightarrow C(x)$ (i.e. the decision condition D applied to a set with "complete extension" is equivalent to the criterion condition C).

Proposition 1: *The unified template is partially correct with respect to the above precondition and postcondition, i.e. if the input data meet the precondition and a particular algorithm instantiated from the template terminates on the input data, then it terminates with the output that meets the postcondition.*

Proof (sketch). Let us represent the template as a flowchart (fig. 1). The flowchart has a single loop. Let us consider

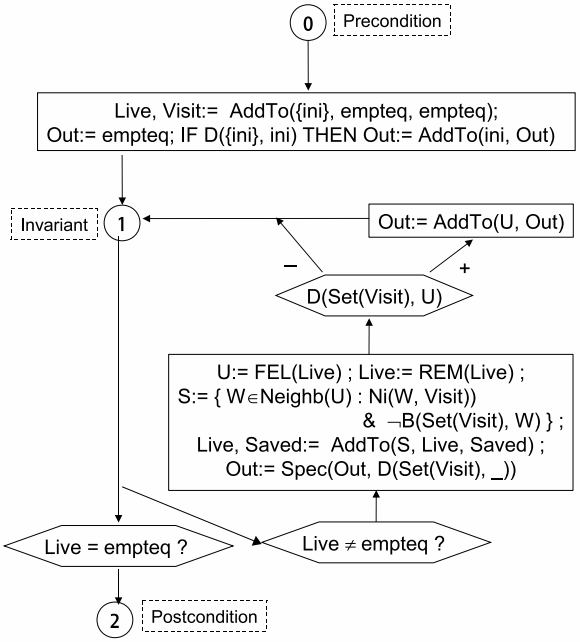

Figure 1. The unified template flowchart

operators grouped into the box before the loop, the loop body guarded by the test $Live \neq empteq?$, and the test $Live = empteq?$ as *big steps* and consider runs in terms the big step semantics (i.e. $next$ construct '•' corresponds to execution of any of these big steps).

Let us adopt as the *invariant* a conjunction of the following clauses.

1) Condition for Out and $Visit$:
   $Out = Spec(Visit,\ \lambda x.D(Set(Visit),\ x))$.
2) Conditions for $Visit$ and $Live$:
   $Live \subseteq Visit$, and for every node $z \in G$, if $Ni(z, Visit)$ and $C(z)$, then z is reachable from $Set(Live)$.
3) Conditions for $Visit$:
   each node $x \in G$ has (at most) single instance in $Visit$, and the set $Set(Visit) \cup Neighb(Set(Visit))$ equals to the set of all nodes that has been generated by the algorithm up to the current moment of time.

We claim that the following safety property *precondition* $\Rightarrow \Box$(at(1) $\Rightarrow$ *invariant*) is valid for all runs of any algorithm instantiated from the template. According to the proof-principles for safety properties [8], it remains to prove by induction that, if the *precondition* holds at the first state of a run, then (at (1) $\Rightarrow$ *invariant*) is valid at every state of the run (i.e. the *invariant* holds at every state, when the control is at (1)).

Assume that the *precondition* holds at the first state of a run and let's proceed by induction. Induction bases: condition (at(1) $\Rightarrow$ *invariant*) holds at the first state of the run, since the control in this state is at the point (0). The most interesting case of the induction step is as follows: assume that *invariant* holds at some state of the run, and that the control is at the point (1); then (at(1) $\Rightarrow$ *invariant*) holds at the next state of the run, since at the next state of the run either the control is at the point (2) and we do not care about the *invariant*, or the control is at the point (1) and the *invariant* holds by construction.

Let us remark that (at(1) $\wedge$ $Live = empteq$) $\Rightarrow$ • at(2) and (at(1) $\wedge$ *invariant* $\wedge$ $Live = empteq$) $\Rightarrow$ *postcondition* hold at every state. Combining these two facts with the proven safety property, one can conclude that *precondition* $\Rightarrow \Box$ (at(2) $\Rightarrow$ *postcondition*) is valid on every run, i.e. that the template is partially correct with respect to the *precondition* and the *postcondition*. ■

Proposition 2: *If the input graph is finite then the unified template eventually terminates, i.e. if the every particular algorithm instantiated from the template always halts traversing the graph after a finite number of steps.*

Proof (sketch). Let N be the number of nodes in G and *Checked* be a new variable of $Node$ type. Let us add the following two assignments to the template: the first one "*Checked* := $\emptyset$" before the loop, and the second one "*Checked* := *Checked* $\cup$ *U*" into the loop body. This modification does not change behavior of the algorithm, since the new variable does not influence any test.

Then let us observe that a conjunction of the following clauses

- $Checked \subseteq Set(Visit)$,
- $Checked \cap Set(Live) = \emptyset$,
- $Set(Live) \subseteq Set(Visit)$,
- $Set(Visit)$ is a set of Node

is an invariant of the loop of the modified template. Hence the following value $(N - |Checked|)$ decreases after each valid loop iteration, but can not become negative. It implies (according to the proof-principals for liveness properties that the algorithm always terminates (after at most N iterations of the loop). ■

The above two propositions imply the following theorem.

Theorem 1: *If the boundary, decision and criterion conditions B, D and C meet the precondition, and the virtual graph G for traversing is finite, then every particular algorithm instantiated from the template terminates after $O(|G|)$ iterations of the loop, and upon termination the set $Set(Out)$ will consist of all nodes of the graph G that meet the criterion condition C.*

This theorem can be applied to n-Queen Puzzle and Discrete Knapsack Problem.

In the case of n-QP, the virtual graph G is well-define, and, hence, the requirement of the *precondition* holds. Let us remark that the boundary and the decision conditions do not depend on (sets of) visited nodes; hence they trivially meet requirements 2 – 4 of the *precondition*. The remaining requirement 5 of the *precondition* is straightforward. It means that the *precondition* is valid. At the same time G is finite. So we are in the conditions of the theorem 1, and hence the above algorithm eventually terminates and $Set(Out)$ will comprise all safe queen placements upon the termination.

In the case of DKP, the boundary, the decision and the criterion conditions in section IV-B meet the precondition too. It implies (according to the theorem) that any B&B algorithm for DKP instantiated from our template is totally correct with respect to the following precondition and postcondition.

Precondition:

- a lower bound is computed according to some fixed method that is monotone on X (i.e. can not decrease while X is expanding as a set);
- an upper bound is computed according to some fixed method that is anti-monotone on y (i.e. can not increase while y is extending lexicographically).

Postcondition: $Set(Out)$ consists of all characteristic vectors $(c_1, \ldots c_n) = \arg\max\{P(c_1, \ldots c_n) : L(c_1, \ldots c_n) \leq W\}$.

VI. Conclusion

We have presented in this paper a unified template for backtracking and branch-and-bound algorithm design patterns, specified the template by means of (semiformal) precondition and postcondition, prove (manually) the total correctness of the template, illustrate how to instantiate a particular annotated algorithm from the specified template.

There are several directions for further research.

The first direction is related to formalization of the template and to development of a computer-aided proof in some proof-assistant system or automatic theorem prover.

Another direction is related to semi-formalization of templates for other algorithm design patterns (dynamic programming for instance), their specification and manual proof.

The third direction deals with build-in the template (and other similar templates in the future) into a system of automatic algorithm generation in an educational programming systems.

ACKNOWLEDGMENT

Research is supported by Russian Basic Research Foundation by grant 09-01-00361-a.

REFERENCES

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[2] K. R. Apt, F. S. de Boer, and E.-R. Olderog, *Verification of Sequential and Concurrent Programs*, 3rd ed. Springer, 2009.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. The MIT Press, 2009.

[4] D. Gries, *The Science of Programming*. Springer, 1987.

[5] S. W. Golomb and L. D. Baumert, *Backtrack Programming*. Journal of ACM, 12(4), 1965, pp.516-524.

[6] A. H. Land and A. G. Doig, *An automatic method of solving discrete programming problems*. Econometrica, 28(3), 1960, pp.497-520.

[7] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer, 1992.

[8] Z. Manna and A. Pnueli, *The Temporal Verification of Reactive Systems: Safety*. Springer, 1995.

[9] V. A. Nepomniaschy, *Verification of finite iterations over collections of variable data structures*. Cybernetics and System Analysis, 43(3), 2007, pp. 341-352.