Solving "How to Clear a Block" with CONSTRUCTIVE *MATCHING* methodology

Marta Franova, Yves Kodratoff
CNRS & Universite Paris Sud,
LRI, Bat. 490,91405 Orsay, France

Abstract[1]

Constructive Matching is a methodology for proving by induction **Specifications Theorems** (ST), i.e., theorems of the form $\forall x\ \exists z\ (P(x) \Rightarrow Q(x,z))$. ST formalize the problem of constructing a program specified by *given an input vector x, an input condition P, find an output vector z verifying Q(x,z), whenever P(x) holds.*

Till now, we have applied our method to the so-called *Constructible Domains*, for which one disposes of a unique way of building any input and output data. The goal of this paper is to enlarge our methodology to well-founded domains which are not constructible. As a simple example, Manna and Waldinger's "How to Clear a Block" problem[2] is solved. As opposed to theirs, our method **does** consider strategic aspects of program derivation from a formal specification.

1 Introduction

In [Franova, 85] we have developed a methodology, called *Constructive Matching* (*CM*), for proving by induction Specification Theorems, i.e., theorems of the form $\forall x\ \exists z\ (P(x) \Rightarrow Q(x,z))$. These theorems formalize program synthesis as follows. Given input vector x verifying the input condition P(x), find an output vector z such that the input-output relation Q(x,z) holds. Practical reasons, such as the availability of examples and implementation limitations, led us to restrict our method to particular inductively constructed sets, called *Constructible Domains* (CD). These are specified by their constructors, enabling new elements to be built from old ones and, possibly, from elements of an already defined domain. In CD each element has a unique representation in terms of constructors. This amounts to saying that we do not allow relations among constructors.

E.g., the set of natural numbers ($\mathcal{NAT}$) can be considered as a CD with constructors **0** and **s**. This gives us the domain $\mathcal{NAT}$ = {0, s(0),s(s(0)), ...} which may be identified with the familiar sequence of integers { 0, 1, 2, ... }. Each element x of $\mathcal{NAT}$ is either **0** or can be written as s(y), where y is an element of $\mathcal{NAT}$. In the following, we shall call s(y), a representative of x, and y a subrepresentative of x.

More generally, we call y a **subrepresentative** of x if it has the form x = Constructor(y).

Moreover, in CD, we have considered mainly recursive[3] definitions given with respect to constructors.

For instance, the basic form of such a recursive definitions for a function f defined on the natural numbers, is

$$f(z) = \begin{cases} z_0, & \text{if } z = 0 \\ g(a,f(a)), & \text{if } z = s(a)\ \&\ H(a) \end{cases}$$

where z_0 is a constant, g is a suitable recursive function and H is a suitable recursive predicate. Note that this definition introduces an existentially quantified variable 'a', which is a subrepresentative of z.

Therefore, no actual new variable has been introduced.

Another typical feature of CD is that definitions do not allow the introduction of existentially quantified variables (EQV) other than subrepresentatives of others. As we illustrate in section 4, the introduction of new EQV is typical of non CD.

For instance, let us consider a predicate P, the recursive case of which is defined by the axiom P(f(u,v),y), If P(u,y)&Q(v), where f is not a constructor. Then, this kind of axiom is not admissible in CD, since it contains EQV[4] u and v. Note that the *put-table-clear* axiom (section 4.3) is of this form.

In order to stress the difference between EQV met in the theorem, and those met in definitions, we shall call the latter ones **unspecified variables** (because they are indeed "unspecified" in the definition).

In general, our method is able to prove a given theorem only if the well-founded order necessary for proving this theorem is expressed in the axioms.

For instance, let us define the addition of two natural numbers by:

$$u+v = \begin{cases} u, & \text{if } v = 0 \\ s(u+p(v)), & \text{if } v > 0 \end{cases}$$

This definition calls recursively the predecessor (p) of given number v. Therefore, the well-founded order determined by this definition allows us to perform only proofs that require one-step induction, i.e., in order to prove, say, $\forall x\ F(x)$, we have to prove F(0) and $F(p(n)) \Rightarrow F(n)$. If $\forall x\ F(x)$ requires other kinds of well-founded orderings, for instance, with respect to the quotient of n when divided by a number m such that $s(0) < m < n$, and if in the theory none of the functions or predicates call such an element, then our current method fails (see details in [Franova, 91a]).

Moreover, if a recursive definition is of the form

$$f(x) = h(x,f(t))$$

we assume that there is a well-founded ordering in which t is smaller than x. We have not yet studied the problem of finding such a well-founded ordering. Presently, we propose that the user uses the results of Boyer and Moore [79] in order to check that his definitions are correct.

[1] This research has been partly supported by the *Programme de Recherches Coordonées en Intelligence Artificielle* from the French *Ministère de la Recherche et de la Technologie*. It is a shortened version of [Franova and Kodratoff, 91a].

[2] Using this problem, Manna and Waldinger show the interest of the deductive approach for the synthesis of imperative programs that may alter data structures or produce other side effects, and illustrate how program synthesis can be carried over to the planning domain. Our goal is the same as theirs, i.e., illustrating that our methodology can be also carried over to planning.

[3] D. Barstow made available to us an example for which we were unable to find a recursive definition of one of the predicates involved in ST. Our method cannot be applied to that kind of problem.

[4] In fact, the axiom P(f(u,v),y), if P(u,y) & Q(v) should correctly be written as P(x,y), if $\exists u\ \exists v\ (x=f(u,v)\ \&\ P(u,y)\ \&\ Q(v))$.

In this paper we enlarge our methodology to well-founded domains (WFD) that are not necessarily constructed with a help of constructors, and which may introduce EQV in the definitions. We explain in [Franova, 88a] that finding a strategy for a proof of an atomic formula is one of basic problems of mechanizing inductive proofs. We therefore limit ourselves here only to this particular problem, even if the simple example we solve here (Manna&Waldinger's "How to Clear a Block" problem [87]), illustrates our overall methodology.

2 CM-methodology

One of the main differences of our method, in comparison with other approaches in inductive theorem proving [Boyer and Moore, 79; Bundy *et al,* 90] is actually very deep since it takes place in the basic step of any theorem proving methodology, viz. in the way atomic formulae are proven. In [Franova, 88a] we have shown the consequences of the choices done at such a low level on the way subproblems are generated during the course of a complete proof.

Classical methods for proving by induction atomic formula can be classified as simplification[1] (or rewrite) methods, i.e., they attempt to transform the atomic formula into simpler and simpler form, until the formula TRUE is reached.

Our method for proving atomic formulae can rather be qualified as a "complication" method, stressing so that we rather progressively build more and more large sets of constraints describing the condition at which the formula is TRUE. The proof is completed when these conditions are proven to be implied by those of the problem.

We call our way of proving an atomic formula a *Constructive Matching* formula construction, or *CM*-formula construction.

Let us give a brief motivation for this name. *If F is an n-ptace predicate symbol and* $t_1 \ldots t_n$ *are terms, then* $F(t_1 \ldots tj$ *is an atomic formula.* This definition of an atomic formula shows that an atomic formula is "constructed", or "build up", from a predicate name and terms in the following manner: We take an n-place symbol F providing the syntactical scheme F(_ , _ . . . . _) , where "_" represent empty positions (or arguments) to be filled up by concrete terms, so that finally $F(t_1 \ldots t_n$ is obtained. In classical thinking, this process of "filling up" empty argument places is mentally performed in one step, i.e., we start from F(_ ,..., _) and reach immediately $F(t_1 \ldots t_n)$.

As opposed to this one step operation, we consider a piece-wise construction of an atomic formula. We start with the syntactical scheme F(_ , ..., _. We then take the first term t_1 and we fill up the first empty argument of our scheme by this term, so we have $F(t_1, \_ \ldots \_)$. Then, we fill up the first empty argument in the last scheme by t_2, obtaining $F(t_1, t_2, \ldots, \_)$, etc.

We thus construct (purely syntactically) *in* n steps Che formula F(t₁...»Q. However, in theorem proving, we need to speak of the validity of a given formula in a theory *T* made from the axioms. This is why we will consider axioms defining the predicate F. These axioms allow us to change the above syntactical construction into a construction which, if successfully performed, provides a proof for $F(t_1 \ldots, t_n)$.

ritamHW

[5] [Beyer and Moore, 79] is an example of a simplification method.

Let us show briefly how axioms are involved in the process of our construction of an atomic formula $F(t_1,t_2)$, created from a predicate F and two terms t_1 and t_2, i.e., n = 2. The order of filling up arguments depends now on given definitions. For simplicity, we suppose here that definitions involved indicate that t_1 has to be filled up first. The *CM*-formula construction can then be briefly described as follows.

We start by building an abstract formula $F(t_1,\xi)$ with an abstract argument ξ, i.e., instead of considering an empty argument place, we take a variable which has a special character since it does not occur in $F(t_1,t_2)$, and moreover it represents (or, equivalently, is an abstraction of) all terms which can fill up the empty argument of the scheme $F(t_1,\_)$ in order to obtain an atomic formula. Therefore, we call it an abstract argument in order to avoid confusing it with the quantified variables of the theorem.

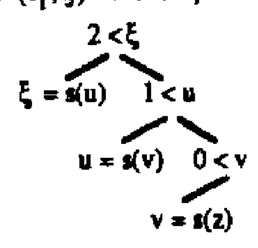
Let us give a very simple example which shows also a difference with the simplification approach, by proving the formula s(s(0)) < s(s(s(s(0)))) (written 2<4, for brevity), and using the following recursive definition of the predicate <:

(1) 0 < n, if n = s(b)
(2) s(m) < n, if n = s(b) & m < b

The definition of "<" is recursive with respect to the first argument. This is why, in the formula 2<4, the second argument (i.e., non-recursive) is replaced by an abstract argument ξ. Up to now, we have constructed purely syntactically the formula 2<ξ, without considering yet the validity of this formula.

Now, the definition of F provides conditions for the validity of the formula $F(t_1,\xi)$. Let us denote by *C* the set of all ξ for which $F(t_1,\xi)$ is true, i.e., $C = \{\xi \mid F(t_1,\xi) \text{ is true}\}$.

Using the definition of the predicate "<" we see that any ξ that satisfies the formula 2<ξ must be of the form ξ=s(u), where 1<u, therefore, *u must be of the form s(v) with 0<v,* and therefore, using (1), v must be of the form s(z). This yields the final form of ξ and so the class of of all ξ that satisfy the formula 2<ξ is



$C = \{\xi$ / there exists z such that $\xi = s(s(s(z)))\}$. Note that we obtain *C* by unfolding [Burstall and Darlington, 77] the formulae 2<ξ, 1<u, 0<v. For each of them we obtain conditions, that are collected and combined. The final step is the withdrawal of the new "abstract arguments" u, v which are recognized as being useless. This leaves ξ expressed in relation with a new variable-argument z.

In other words, we solve our problem in a top-down manner, pick up the final leaf (v=s(z)), carry this bottom leaf up to the root of the tree. During this last step, the final leaf is applied to all other leaves of the tree. The solution obtained (in case of success) is given by the set of independent leaves in which the final leaf has been applied.

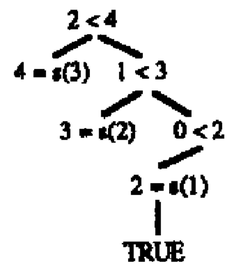
In our example, final leaf is v=s(z). It is applied successively to u=s(v) yielding u=s(s(z)), then to ξ=s(u), yielding ξ=s(s(s(z))). Therefore we have that the solution of our problem 2<ξ is ξ=s(s(s(z))).

We are then left with checking if the replacement of ξ by t_2 preserves the validity of $F(t_1,\xi)$, i.e., we have to check whether $t_2 \in C$.

In our example, we have to check whether 4 belongs to *C*. In fact, it does, since 4 = s(s(s(s(0)))), i.e., z = s(0).

The same problem is tackled by the simplification approach in a different manner:

Applying (2) to 2<4 leads to two subproblems. Firstly, it is necessary to show that there is an element b such that 4=s(b). Here, we succeed since b=3. Then it is necessary to prove 1<3. In this step, our original problem, proving the formula 2<4 is replaced by another equivalent problem: Prove 1<3. One therefore "forgets" the original problem. Applying (2) to 1<3 leads again



to two similar subproblems. Firstly, it is necessary to show that there is an element b such that 3=a(b}. Mere, we succeed since b-2 Then it is necessary to prove 0<2. Here, once again we "forget* the formula 1<3 and we concentrate our effort to the formula 0<2 Using (1) this simplifies to TRUE, because them is an element b (b=1) such that 2=(b). This prows our original formula.

This simplification approach proves the formula in a top-down manner, but the final leaf is TRUE in the case of success. Therefore, as opposed to our method, there are no final bottom-up steps in the simplification approach.

It may seem that our "constructive" method makes the proof more complicated (as compared to the simplification methods) without anything to gain. Recall, however, that simplification procedures have been developed for theorems without EQV. Therefore, our method which is suitable for specification theorems, is more powerful, even if it may seem more awkward and non-useful for theorems with universally quantified variables only. As compared to classical simplification thinking, it may seem also more artificial, because, by our method, we may generate an existentially quantified lemma when proving a universally quantified theorem. The simplification methods are built in such a manner that all the subproblems generated are universally quantified, while this restriction is not necessary to our CM-methodology,

Program synthesis methodologies ([Manna and Waldinger, 80; Kodratoff and Picard, 83; Bibel and HOrnig, 84; Dershowitz, 85; Smith, 85; Perdrix, 86; Biundo, 88]) do not consider the problem of strategy for proving an atomic formula as the main problem. In fact, these methods take the whole specification, say $Q_1(x,z)$ & ... & $Q_n(x,z)$, where $Q_1 \ldots, Q_a$ are literals, and perform transformations on this complex formula*. As opposed to such a treatment of a given specification, our method deals firstly[7] with $Q_1(x,z)$, performing the CA-formula construction it finds conditions for a validity of this formula. After this step has been completed, arid assuming the conditions obtained, our method starts taking care of $Q_2(x,z)$, and so on, until the last literal Qn(X,Z) is treated

Thus, as a summary, let us state that, in order to prove an atomic formula $F(t_1,t_2)$, created from a predicate F and two terms t_1 and t_2, we start by building an abstract formula $\mathbf{F(t_1,\xi)}$ with an abstract argument ξ. The definition of F provides conditions for the validity of the formula $\mathbf{F(t_1,\xi)}$. Let us denote by *C* the set of all ξ for which $\mathbf{F(t_1,\xi)}$ is true, i.e., $C = \{\xi \mid \mathbf{F(t_1,\xi)}$ is true$\}$. We are then left with checking if the replacement of ξ by t_2 preserves the validity of $\mathbf{F(t_1,\xi)}$, i.e., we have to check whether $t_2 \in C$

3 CM-formula construction for WFD

In the previous section we pointed out that the main reason for performing the CW-formula construction is its suitability for proving specification theorems. We also described the way to proceed when a given specification is a conjunction of atomic formulae. Therefore, in this section, we shall concentrate on a specification theorem[8] $\forall x\, \exists z\, Q(x,z)$, where $Q(x,z)$ is an atomic formula.

Let us denote by ST_0 this theorem and by σ_p the substitution $\{x \leftarrow p\}$, where p is smaller than x. Using the general induction principle and x being the induction variable, we have to prove $\{\sigma_p(\exists z\, Q(x,z))\} \Rightarrow \exists z\, Q(x,z)$.

For this general induction principle scheme, we do not have a division into a base step and an induction step. From the computational point of view, however, it is interesting to "simulate" a base step for ST_0, since we then obtain the non-recursive parts of the desired programs.

3.1 Base step (a general outline)

One way to simulate a base step is done by using the tautology $Q(x,t) \Rightarrow Q(x,t)$ as explained and exemplified in [Franova and Kodratoff, 91a].

Let us consider now the induction step, i.e., we try to find out recursive parts of the program.

3.2 Induction step solutions (a general outline)

During this step we shall perform the *CM*-formula construction presented in an intuitive way in section 2. For simplicity, and in order to keep the notation of section 2, Q(x,z) will be called $F(t_1,t_2)$[9]. We suppose that F is defined in *T* recursively (at least) with respect to the first argument.

As pointed out in section 2, we start by building an abstract formula $F(t_1,\xi)$ with an abstract argument ξ. Then

Step 1: We ***evaluate*** the term t_1, i.e., we apply to t_1 axioms defining functions occurring in t_1 until no further axioms can be applied. For brevity, we suppose here that t_1 is already evaluated, and that the evaluations yielded no condition.

Step 2: We ***evaluate*** $F(t_1,\xi)$ in order to ***find conditions for the validity*** of the formula $F(t_1,\xi)$.

With respect to the recursive definition of F this step gives a composition of the recursive call and some formula, say G. We have $F(t_1,\xi)$ **if** comp(G,F(t,ξ)).

Here, it may happen that t in the recursive call is expressed in terms of new EQV[10] that do not appear in ST_0. For instance, later we show that constructing the formula Clear(ξ,b) (using the *put-table-clear* axiom from section 4.2) leads to

Clear(ξ,b) if ξ = put'(w,x,table) & On(w,x,b) & Clear(w,x).

This formula says that Clear(ξ,b) is valid only if there is a state w and a block x such that ξ can be written as put'(w,x,table) and moreover if On(w,x,b) & Clear(w,x) holds. Thus, x and w are existentially quantified, i.e., x and w are unspecified variables. The presence of such variables in a proof is not admissible, because it makes the proof to be a nonconstructive one. Therefore, during an actual proof we have to express unspecified variables in terms of variables given in ST_0. Whenever we express a variable in terms of variables given in ST_0, we say that we **concretize** this variable, or, equivalently, we say that we try to **make** this variable **more explicit**. The following step is necessary only when such unspecified variables appear.

Step 2*: We try to ***concretize*** unspecified variables until the recursive call formula contains variables of ST_0 so that the application of the induction hypothesis is possible.

We will try to concretize unspecified variables in F(t,ξ) exploring where the atomic formulae of comp(G,F(t,ξ)) may come from. Thus, we apply our *CM*-formula construction to

See (Franova. 91c] for mote detail* about differences between our method and the other program synthesis methods.

[7] There are heuristics which suitably order literals in a given specification, For instance, if x is an input and z is an output, then the specification ordered(z) & permui(x,z) is reordered to permul(x,z) &. ordered(z) Moreover, in such a reordered specification, $Q_1(x,z)$ must be an atomic formula.

[8] For simplicity, we assume here that the input condition P(x) is TRUE.

[9] Thus, we assume here that Q(x,z) is created from a binary predicate F and the terms t_1 and t_2.

[10] Remember that we call such EQV "unspecified variables" and that such variables are not possible for CD.

the atomic formulae of comp(G,F(t,ξ)). Clearly, in general, heuristics have to be developed here to choose the order in which the formulae from comp(G,F(t,ξ)) are examined. In section 4 we will follow one possible heuristic: The priority of formulae with a larger number of variables of ST_0.

This concretization of unspecified variables changes F(t,ξ) to a new formula, say F(t',ξ).

Step 3: We now try to ***apply induction*** hypotheses in order to ***concretize***[11] ξ.

Recall that $\sigma_p = \{ x \leftarrow p \}$, where p is smaller than x.
The generic form of induction hypotheses (IH) is

$$\forall u\ \sigma_p F(t_1(x,u),t_2(x,u)).$$

The quantification here is due to a possible presence of universally quantified variables other than x in ST_0. Thus, if ST_0 contains no other universally quantified variables, the form of IH is $\sigma_p F(t_1(x),t_2(x))$. Attempting to apply IH leads to a comparison between F(t',ξ) and $\forall u\ \sigma_p F(t_1(x,u),t_2(x,u))$, which, in general, yields the transformational problem represented by the equations

$$\exists u\ (\ t' = \sigma_p t_1(x,u)\ )\ \&\ (\ \xi = \sigma_p t_2(x,u)\ ).$$

The solution of this problem allows the replacement of ξ by the more concrete element $\sigma_p t_2(x,u)$. Let us denote this element by α. Therefore, the elements of the class for which the theorem is true, *C*, are represented by α.

Step 4: Our goal is now to verify if t_2 belongs to *C*, i.e., we have to check if we can **replace** α by t_2.

This is equivalent to performing a unification of α and t_2 modulo the theory *T*. Our *CM*-term construction performs this operation (see [Franova, 88b]).

Step 5: We have to perform final simplifications.

The reader can now see that our formula construction is an algorithm which mechanizes a proof of an atomic formula, because a successfully completed construction provides a proof. On the other hand, this construction is an algorithm, because we know what we want to obtain and how to obtain it.

- Clearly, in order to find conditions for the validity of $F(t_1,\xi)$ we evaluate $F(t_1,\xi)$ using the definition of F in the same way as the unfold operation [Burstall and Darlington, 77].
- We know then that we have to apply induction hypotheses, so we generate equations comparing IH and the previously obtained expansion.
- The success of the last equation solving problem allows the concretization of ξ by a more concrete α and allows at least the most trivial simplification, which is here eliminating recursion formulae from the obtained expansion.
- We have then to replace α by t_2, so we generate an equation between α and t_2.

4 Example: How to Clear a Block (HCB)

Here, a plan for the problem[12] is specified as follows. The problem is to clear a given block, where we are not told whether the block is already clear or, if it is known not to be clear, how many blocks are above it. We adopt here the plan theory developed by Manna and Waldinger [87] (further referred to as the row-plan theory) for describing situational logic events in terms of classical logics, and their notation.

4.1 Notations

For a given blocks a, u and v
clear (a) is true if the block a is clear
on (u, v) is true if the block u is on the object v
hat (a) is the block directly on a, if it exists
put(u,v) is the action which places block u on top of v

In situational logic we have to consider the value of a function or a predicate with respect to a state, i.e., we have to introduce an explicit state argument w for them. For example, for the predicate clear and the function hat we have

Clear(w,a) is true if the block a *is* clear in state w
hat'(w,a) is the block on top of the block a in state w

Actions are represented as functions that yield states; for example put'(w,u,v) is the state obtained from state w by putting block u on object v.

4.2 Axioms for mw-plan theory

THE FUNCTION ":"

If s is a state and e an object, then s;e denotes the object designated by e in state s.

To any n-ary function symbol f a new n+1 -ary symbol f is associated with the property

$$w{:}f(u_1,\ldots,u_n) = f'(w,w{:}u_1,\ \ldots,\ w{:}u_n) \quad \textit{(object linkage)}$$

for example, a fixed block w:hat(u) can be expressed equivalenuy by hat'(w,w:u).

THE RELATION"""

This relation is analogous to ":", but the relation :: *is for* predicates. If s is a state and e is a proposition, then s::e is a proposition denoting the truth-value designated by e in state s. E.g., s::clear(d) is true if the block s:d is clear in state s.

Analogously to the *object linkage,* the *propositioned iinkage* linkage axiom is introduced. To any n-ary predicate symbol r a new n+l-ary symbol R is associated with the property

$$w{::}r(u_1,\ldots,u_n) = R(w,w;u_1,\ldots,w.u_n) \quad \textit{(prepositional Linkage)}$$

for example, s::clear(d) = Clear(s,s:d), i.e., s:xlear(d) is true if the block s:d is clear in state s.

THE EXECUTION FUNCTION ";"

If s is a state and p a plan, s;p denotes the state obtained by executing plan p in state s. E.g., s;put(a,d) is the state obtained by putting clock a on object d in state s.

Analogously to the above link*age* axioms, the *plan linkage* linkage axiom is introduced. To any n-ary plan symbol g a new n+l-ary symbol g' is associated with the property

$$w;g(u_1\ldots,u_n) = g'(w,w{:}U_1,\ \ldots,\ w{:}u_n) \quad \textit{(planLinkage)}$$

for example, w;put(u,v) = put'(w,w:u,w:v).

The *empty plan* A is taken to be a right *identity* under the execution function. This *is* formalized by the axiom.

$$w\ ;\ A = w \quad \textit{(empty plan)}$$

There are objects that do not depend on states considered. For instance, the constant table always denotes the same object. These objects are called rigid designators, i.e., an object u is a *rigid designator,* if for all states w

$$w{:}u = u \quad \textit{(rigid designator)}$$

THE PLAN COMPOSITION FUNCTION "ii"

This notion of composing plans is introduced in the following way. If p_1 and p_2 are plans, $p_1;;p_2$, is the composition of p. and p, where it is understood that P_1 is executed first ana then only p_2 is executed. This is expressed by the *pirn composition* axiom

$$W;(P_1;;P_2) = (W;P_1);P_2 \quad \textit{(plan composition)}$$

[11] Thus in thii step we try to concretize the abstract argument by expressing it in terms of variablesof the theorem under consideration and/or of induction hypotheses corresponding to this theorem. It may happen that a given theory allows a direct replacement of E by lomething more explicit provided some condition C if verified. If C is TRUE and we succeed in performing steps 4 and 5 of our procedure, then it meant that the given theorem is provable without the use of induction. If C if not TRUE, then we obtain a conditional solution which, in the case of specification theorems, provides non-recursive pans of programs (as illustrated in section 4).

It is assumed that we are in a world of bocks in which objects are a table and blocks. These blocks are all the same size, so that only one block can fit directly on top of another. It it also assumed that the robot arm may lift only one clock at a time.

for all states w and plans p_1 and p_2. For simplicity, the distinction between the composition function ;; and the execution function ; is ignored. We will write ; for both and rely on context to make the meaning clear.
This composition is assumed to be associative.

$$(p_1;p_2);p_3 = p_1;(p_2;p_3) \qquad \textit{(associativity)}$$

The *empty plan* Λ is taken to be the *identity* under composition, i.e., for all plans p

$$\Lambda;p = p;\Lambda = p \qquad \textit{(identity)}$$

4.3 Axioms for the blocks world

Facts about the block world and effects of actions are expressed as plan theory axioms. For simplicity, sort conditions such as state(w) are omitted. Variables are understood to be universally quantified.

If not Clear(w,y) then On(w,hat'(w,y),y) *(hat)*

The *hat* axiom describes the following situation: If (in the state w) the block y is not clear then there is a block directly on y.

If **Clear(w,x)** *(put-table-on)*
then **On(put'(w,x,table),x,table)**

The *put-table-on* axiom allows to put a block x (in the state w) on the table whenever x is a clear block, i.e., if there is no block directly on x.

If **On(w,x,y) & Clear(w,x)** *(put-table-clear)*
then **Clear(put'(w,x,table),y)**

The *put-table-clear* axiom describes the following situation: If (in the state w) the block x is directly on the block y and if x is a clear block, then y becomes a clear block when we put x on the table.

Even if the given axioms could suggest that we might define constructors **hat** (for objects) and **put** (for actions), the given domain is nonconstructible, because the given axioms introduce new EQV (see section 1), such as *w* and *x*.

4.4 Specification Theorem for HCB

The problem to clear a given block is expressed as:

$$\forall s_0 \forall b\ \exists z_1\ \mathrm{Clear}(s_0;z_1,b).$$

As in [Manna and Waldinger, 87], we denote the Skolem function corresponding to this ST by makeclear, i.e., we try to find a function makeclear such that $z_1 = \mathrm{makeclear}(s_0,b)$.

4.4.1 Choice of induction variable

The variable b is chosen as the induction variable.

4.4.2 Generating IH

Using the general induction principle scheme, we generate the induction hypothesis in the state s_0

If u ∠ b then there is z such that
Clear(s_0;z,(s_0;z):u) (IH)

Here ∠ is an unspecified yet well-founded ordering. During the actual proof, we shall need to assert that the objects are indeed in such a relation, thus defining the well-founded ordering, on the fly, so to speak. If the recursive functions we start from are really computable, they should be associated to an existing well-founded ordering, which is exactly the one we are thus discovering.

Now, we will follow the algorithm described in section 3. To F in the scheme corresponds Clear, to t_1 corresponds b, to t_2 corresponds $s_0;z_1$.

4.4.3 $\mathcal{CM}$-construction of Clear(s_0;z_1,b)

4.4.3.1 Base step solutions

This means that we try to construct the formula $\mathrm{Clear}(s_0;z_1,b)$ (i.e., find a value z_1 for which the last formula is true) using the tautology $\mathrm{Clear}(\xi,b) \Rightarrow \mathrm{Clear}(\xi,b)$. The part of code we extract in this step [Franova and Kodratoff, 91a] is

$$\mathrm{makeclear}(s_0,b) = \Lambda, \quad \text{if} \quad \mathrm{Clear}(s_0,b).$$

4.4.3.2 Induction step solutions

We try to construct the formula $\mathrm{Clear}(s_0;z_1,b)$ (i.e., find a value z_1 for which the last formula is true) using IH.

Step 1: The term b is already evaluated.

Step 2: The definition of Clear gives conditions for the validity of the formula

$$\mathrm{Clear}(\xi,b) \qquad (1)$$

The only axiom which leads to a formula of the form Clear(...) is *put-table-clear*. In order to be able to apply it, we have to compare (1) and Clear(put'(w,x,table),y), where w, x, y are universally quantified. Comparing (1) and Clear(put'(w,x,table),y) leads to the equation solving

$$\xi = \mathrm{put'}(w,x,\mathrm{table}), \quad b = y.$$

Let us denote by PrecondAx the precondition of the *put-table-clear* axiom in which the variable y is replaced by b, i.e., the formula

$$\mathrm{On}(w,x,b)\&\mathrm{Clear}(w,x). \qquad (2)$$

Thus, using the *put-table-clear* axiom, the class of all ξ for which Clear(ξ,b) holds is determined by

$$C = \{\ \xi \mid \xi = \mathrm{put'}(w,x,\mathrm{table})\ \&\ \mathrm{PrecondAx}\ \}. \qquad (3)$$

This reads: Any element ξ satisfying the formula Clear(ξ,b) must have the form put'(w,x,table), for any w and x satisfying (2). Nevertheless, variables w, x are still unspecified. Next steps will deal with that.

Step 2*: Let us consider PrecondAx, i.e., formula (2). As we have explained in section 3, we will try to concretize w and x in (2) exploring where the atomic formulae of PrecondAx may come from. Thus, we have to apply our *CM*-formula construction to (some of) the atomic formulae of (2). The formula On(w,x,b) contains a known value, namely b, this is why this formula is first examined. The application of *put-table-on* is not possible, if we suppose that we cannot identify the table and an object, which is here the element b. The only axiom we can apply is therefore the *hat* axiom. The application of this axiom requires comparing On(w,x,b) and On(v,hat'(v,q),q). The solution we obtain is

$$v \leftarrow w,\ q \leftarrow b,\ x \leftarrow \mathrm{hat'}(w,b).$$

The precondition of the *hat* axiom reads then not(Clear(w,b)). (3) changes here to

$$C = \{\ \xi \mid \xi = \mathrm{put'}(w,\mathrm{hat'}(w,b),\mathrm{table})\ \&\ \mathrm{Clear}(w,\mathrm{hat'}(w,b))\ \&\ \mathrm{not}(\mathrm{Clear}(w,b))\ \}. \qquad (4)$$

Next step will concretize w.

Step 3: As we have mentioned already in footnote 11, this step consists of two subproblems:

Step 3.1: Trivial Transformations

In [Franova and Kodratoff, 91a] we explain in detail that this step succeeds to find the following conditional non-recursive part of the program we want to synthesize:

$$\mathrm{makeclear}(s_0,b) = \mathrm{put}(\mathrm{hat}(b),\mathrm{table}) \quad \text{if} \quad \mathrm{Clear}(s_0,\mathrm{hat'}(s_0,b))\ \&\ \mathrm{not}(\mathrm{Clear}(s_0,b))$$

Step 3.2: Non - trivial Transformations

We try to apply induction hypotheses to (4). Comparing IH and formulae in (4) we can see that IH (with $z = \mathrm{makeclear}(s_0,u)$) can be compared to the formula Clear(w,hat'(w,b)). This leads to the equations

$$w = s_0;\mathrm{makeclear}(s_0,u), \quad u = \mathrm{hat}(b)$$

However, the application of IH is justified only if we are able to prove that hat(b)∠b.[13] Let us assume that we have means to confirm this relation. This leads to

[13] Some complementary knowledge is necessary to establish this relation. Presently, when we check that such a relation holds during the application of the induction hypothesis, we assume *a priori* that this relation holds, in order to avoid interrupting the theorem proving process. The validity of such an assumption can be verified by the user. We are planning to automatize this process. This problem is

$$w \leftarrow s_0;makeclear(s_0,hat(b)).$$

Using *plan linkage*, we have put'(w,hat'(w,b),table) = w;put(hat(b),table). Finally, the class C changes to

$$C = \{ \xi \mid \xi \approx s_0;makeclear(s_0,hat(b));put(hat(b),table) \ \&\ not(Clear(s_0;makeclear(s_0,hat(b)),b))\}, \quad (5)$$

i.e., we can replace the abstract argument ξ by the more concrete one

$$s_0;makeclear(s_0,hat(b));put(hat(b),table).$$

We denote it, as in the general scheme, by α.

Step 4: Our goal is now to verify if $s_0;z_1$ belongs to C in (5), i.e., we have to check if we can **replace** α by $s_0;z_1$. Trivially, we obtain

$$z_1 \leftarrow makeclear(s_0,hat(b));put(hat(b),table).$$

Step 5: No simplifications are necessary.

In conclusion, we have the program

makeclear(s_0,b) ⇐
  A, if Clear(s_0,b)
  put(hat(b),table), if Clear(s_0,hat'(s_0,b))&not(Clear(s_0,b))
  makeclear(s_0,hat(b));put(hat(b),table).

5 Conclusion

In all our previous work we presented the *CM*-construction of atomic formulae as a method which helps to solve strategic aspects of inductive theorem proving applied to program synthesis. This paper shows that our *Constructive Matching* methodology can be applied and adapted to other well-founded theories, not only to CD, even though the present implementation works only for CD.

Summarizing, wc have illustrated in this paper the following characteristic features of our *CM* methodology:

G It provides a couple (what has to be achieved; what it has to be achieved from) at any step of an inductive proof. This is the main characteristic. Mote that this feature reduces the search space of a proof, the last is the main problem of deductive approach to program synthesis.

G For the induction step, it uses the general induction scheme and a kind of 'forced' application of this scheme (more details are given in section 4).

G If necessary, it applies the Cm-formula construction to formulae obtained as conditions lor the validity of F (see step 2').

CM methodology can be compared to the notion of a plan in Bundy's reconstruction [Bundy, 88] of Boyer&Moore methodology. Our methodology could create one of the most general plans in Bundy's reconstruction, because it comprises most of Bundy's "reconstructed" tactics[14]. A deep analogy between Bundy's approach and our methodology is illustrated by the importance of universally quantified induction hypotheses acknowledged by both approaches [Franova, 85; *Bundy et al., 901*

similar lo verifying the correctness of the given axioms. Besides, the relation hat(b) < b seems to be a reasonable well-founded relation for induction proofs in blocks world problems. However, analogously to constructible domains, for particular problems, how the well-founded relation looks, depends always on the given axioms (or the given theory). In other words, we cannot know in advance the well-founded relation which is lo be used in blocks world problems, since it depends on the available definitions.

For instance, all examples presented in fBundy e*t al.,* 90] are successfully solved by our methodology. This shows that our way of constructing a formula provides a solution for the problem of making possible the application of induction hypotheses, this problem being the topic of Bundy's "rippling-out" tactics. Since we prove also theorems containing existential quantifiers, our approach is clearly more general than Bundy's.

Moreover, a "rational reconstruction" of our methodology would allow the introduction of existential quantifiers into Bundy's improved system, thus recognizing program synthesis as an inductive theorem proving problem, or, in other words, bringing program synthesis back where *it* belongs classically - inductive theorem proving.

In [Franova, 91c] wc describe in detail the strengths and weaknesses of our methodology viewed as a program synthesis methodology.

Acknowledgments

We express our thanks lo an anonymous referee for many constructive critics.

References

JBbel and Horn®, 84] W. Bibel, K. M. Hdmig; *LOPS - A System Based on* a Strategical Approach to Program Synthesis: in A. Biermann, G. Guino.Y. Kodnaioff (ed): *Automatic Program Construction Techniques,* Macmillan Publishing Company, London. 1984,69-91.

[Biundo, 68] S Biundo: Automated synthesis of recursive algorithms as a theorem proving *tool: in* [Kodratoff. 88], *553-558*

(Boyer and Moore, 79] R. S. Boy, J S Moore: *A Computational Logic,* Academic Press, 1979.

[Bundy, 88] A Bundy: The use of Explicit Plans to Guide Inductive Proofs: in E. Lusk, R. Overbook, (ed): *9th International Conference on Automated Deduction]* LNCS 310, Springer-Verlafl. *Berlin, 1988,* 111-120.

Bundy *et al.,* 90] A. Bundy, F. van Harmelen, A. Smaill, A. Ireland: Extensions to the Rippling-Out Tactic for Guiding Inductive Proofs; in M E. Siickel. {ed.}: (ed) *International* Conference on *Automated Deduction;* Proceedings, Lecture Notes in Artificial Intelligence No. 449, Springer-Veriag, 1990, 132-146

[Burstall and Darlington, 77] R M. Burstall, J.Darlington: *A* trans *formation system* for developing recursive programs; *J ACM* 24,1, January, 1977.44-67.

[Dershowitz, 85] N. Dershowitz: Synthesis by Completion; in [Joshi, 85], 208-214.

[Franova and Kodratoff, 91a] M. Franova, Y. Kodratoff: Solving "How to Clear a Block" with CONSTRUCTIVE MATCHING methodology, extended version of this paper, Rapport de *Recherche* LRI., July, 1991.

[Franova and Kodratoff, 91b] M. Franova, Y. Kodratoff: Program Synthesis is Theorem Proving; Rapport de Recherche LR. I., July, 1991.

[Franova, 85] M Franova: CMstrategy : A Methodology for Inductive Theorem Proving or Constructive Well-Generalized Proofs; in [Joshi, 85], 1214-1220

[Franova, 88a] M. Franova: Fundamentals for a *new methodology* for inductive theorem proving: CM-construction of atomic formulae; in [Kodratoff, 88], 137-141.

[Franova, 88b] M. Franova: Fundamentals of a new methodology for Program Synthesis from Formal Specifications: CM-constructton of atomic formulae; Thesis, Universite Paris-Sud, November, Orsay, France, 1988.

[Franova, 91*a]* M. *Franova:* Generating induction hypotheses by Constructive Matching methodology for Inductive Theorem Proving and Program Synthesis revisited; Rapport de Recherche No.647, LRI, Universite de Paris-Sud, Orsay, France, February, 1991.

[Franova, 91b] M. Franova: Failure analysis in Constructve Matching methodology; Rapport de Recherche LRI., July, 1991.

[Franova, 91c] M. Franova: Constructive Matching methodology for Inductive Theorem Proving and Program Synthesis revisited; RR L.R.I., July, 1991.

[Joshi, 85] A. K. Joshi, (ed): *Proceedings of the Ninth International Joint Conference on Artificial intelligence.* August, Los Angeles, 1985.

[Kodratoff and Picard, *83}* Y. Kodratoff, M. Pcard: Completion de sysiemes de reecriture et synthase de programmes a partir de leurs specifications; Bigre No 35, October, 1983.

[Kodratoff, 88] Y. Kodratoff, (ed): *Proceedtngs of the 8th European Conferance on Arlificial Intelligence,* August 1-5, Pitman, London, United Kingdom, 1988.

[Manna and Waldinger, 80) Z. Manna, R.Waidinger: A Deductive Approach to Program Synthesis; *ACM Transactions on Programming Languages and Systems,* Vol. 2., No.1, January, 1980,90-121.

[Manna and Waldinger, 87] Z Manna, R. Waldinger: How to Clear a Block: A Theory of Plans; *Journal of Automated Reasoning* 3,1987,343-377

[Perdrix, 86] H Perdnx: Program synthesis from specifications; in *ESPRIT'85, Status Report of Continuing* Wort, North-Holland, 1986, 371-385.

[Smith, 85) 0. R. Smith: Top-Down Synthesis of Simple Divide and Conquer Algorithm; *Artificial Intelligence,* vol. 27. no. 1,1985,43-96.