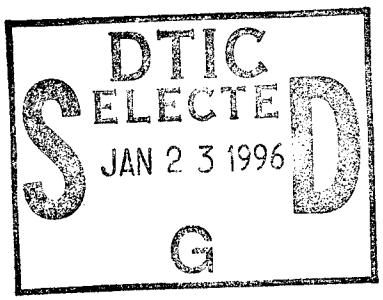
RL-TR-95-205
Final Technical Report
October 1995

PLANNING IN KIDS

CALSPAN-UB Research Center

Dr. Carla Gomes



APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

Rome Laboratory
Air Force Materiel Command
Rome, New York

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS, it will be releasable to the general public, including foreign nations.

RL-TR-95- 205 has been reviewed and is approved for publication.

APPROVED: Karen M. Alguire

KAREN M. ALGUIRE
Project Engineer

FOR THE COMMANDER: John A. Graniero

JOHN A. GRANIERO
Chief Scientist
Command, Control & Communications Directorate

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify Rome Laboratory/ (C3CA), Rome NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	October 1995	Final Feb 94 - Dec 94

4. TITLE AND SUBTITLE

PLANNING IN KIDS

5. FUNDING NUMBERS

C - F30602-93-D-0075, Task 10
PE - 62702F
PR - 5581
TA - 27
WU - PN

6. AUTHOR(S)

Dr. Carla Gomes

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

CALSPAN-UB Research Center
4455 Genesee Street
Buffalo NY 14225

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Rome Laboratory/C3CA
525 Brooks Rd
Rome NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

RL-TR-95-205

11. SUPPLEMENTARY NOTES

Rome Laboratory Project Engineer: Karen M. Alguire/C3CA/(315) 330-4833

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This report gives an overview of KIDS (Kestrel Interactive Development System). We describe a program derived with KIDS to solve the Missionaries and Cannibals problem (MC PROBLEM), a classical AI planning problem. Our interest in this problem arose as the result of some difficulties we experienced trying to use the AI planners O-Plan2 and SIPE-2 to solve it. We were curious as to how one could solve a planning problem such as the MC PROBLEM using KIDS. We also wanted to evaluate how difficult it would be for someone to formalize and derive a program with KIDS. We derived a program for the K-MC PROBLEM that would reach the solution for the 3-MC PROBLEM in 108 msecs (real time) and 0 msec (cpu time). For the same problem, on the same machine, SIPE-2 takes 26.19secs (real time) and 0.15 secs (cpu time). We were able to get the solution for 100 problems, the K-MC PROBLEM (k=1,2,...100), in 225.009 secs (real time) and 11.467 secs (cpu time) using KIDS. SIPE-2 could not solve problems for k > 45 in less than 20 minutes real time.

14. SUBJECT TERMS

Software synthesis, Formal methods, Problem specification, Artificial intelligence, Nonlinear planning systems

15. NUMBER OF PAGES

64

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Contents

1 Introduction

In this paper we give an overview of KIDS (Kestrel Interactive Development System) [Smith 90, Smith 91]. We describe a program derived with KIDS to solve the Missionaries and Cannibals problem. The Missionaries and Cannibals Problem (MC PROBLEM) is a classical AI planning problem. Our interest in this problem arose as the result of some difficulties that we experienced when trying to use the AI planners O-PLAN2 and SIPE-2 to solve it [Gomes 94]. We were curious to see how one could solve a planning problem such as the MC PROBLEM using KIDS. We also wanted to evaluate how difficult it would be for someone to formalize and derive a program with KIDS. We derived a program for the K-MC PROBLEM that would reach the solution for the 3-MC PROBLEM in 108 msecs (real time) and 0 msec (cpu time). For the same problem, on the same machine, SIPE-2 takes 26.19 secs (real time) and 0.15 secs (cpu time). We were able to get the solution (or "no solution")[2] for 100 problems, the K-MC PROBLEMS ($k = 1, 2, \ldots 100$), in KIDS in 225.009 secs (real time) and 11.467 secs (cpu time). SIPE-2 could not solve problems for $k > 45$ in less than 20 minutes real time.

2 About KIDS

KIDS is an framework for the development of programs from formal specifications [Smith 90, Smith 91]. KIDS provides tools for:

- deductive inference
- algorithm design
- expression simplification
- finite differencing
- partial evaluation
- other transformations

KIDS is a program-transformation system. A user of KIDS develops a formal specification into a program by interactively applying a sequence of high-level transformations. KIDS' unique features are the algorithm design tactics (e.g., global search or divide-and-conquer) and its use of a deductive inference component. All the KIDS transformations are correctness preserving. KIDS runs on SUN-4 workstations and it is built on top of REFINE, a commercial knowledge-based programming environment and a high-level language. REFINE language supports (a subset of) first-order logic, set theory, pattern matching, and transformation rules. Refine has its own compiler that generates CommonLisp code.

The steps to follow in order to build a program in KIDS are:

[2]There are no solutions for $k > 3$.

- Develop a *domain theory* to state and reason about the problem - the user defines appropriate types and functions that describe the problem. The user also provides laws that allow high-level reasoning about the defined functions - in particular distributive and monotonicity laws provide most of the laws that are needed to support design and optimization
- Apply a *tactic* - a well-structured but inefficient algorithm is created - the user selects an algorithm design tactic (method) from a menu and applies it to the specification. Existing tactics are:
  - divide-and-conquer
  - global search
    * binary search
    * backtrack
    * branch-and-bound
- *Simplifications, partial evaluation and finite differencing*, and other transformations are applied in order to improve the efficiency of the code
- *Compilation* of the code

KIDS uses a form of deductive inference called ***directed inference*** to reason about the problem specification in order to optimize the code, apply tactics, and derive filters [Smith 82, Smith 87]. Directed inference allows the derivation of sufficient conditions (antecedents) as well as the derivation of necessary conditions (consequents). In directed inference a *source formula (S)* is transformed into a *target formula (T)* bearing a specified relationship (inference direction (*rightarrow*) to the first and given certain *assumptions (A)*. The *rightarrow* is a reflexive and transitive ordering relation between terms.

$$Find\ Some\ (Target)(A \Rightarrow (Source(x_1..,x_m) \rightarrow Target(Y_1,..,Y_M)))$$

Directed inference performs first-order theorem-proving and formula simplification as special cases. The inference direction is one of the following:

- forward inference $\Rightarrow$
- backward inference $\Leftarrow$
- simplification $=$
- derivation of a lower bound $>=$
- derivation of an upper bound $<=$

The inference process involves applying a sequence of transformations to the original form. The transformations are restricted to those that preserve the specified inference direction.

3 Derivation of a Program for the Missionaries and Cannibals Problem in KIDS

3.1 The Missionaries and Cannibals Problem

Three missionaries and three cannibals find themselves on the left bank of a river. The missionaries and cannibals want to cross the river to the other bank but there is only a two-person boat to transport them. Missionaries and cannibals agree to share the boat. However, the missionaries do not trust the cannibals. They want to devise a plan to cross the river in such a way that the number of cannibals on either side of the river never outnumbers the number of missionaries who are on the same side.

A generalization of the problem just described considers k missionaries and k cannibals - we refer to this problem as the K-MC PROBLEM. When $k > 3$ the K-MC PROBLEM does not have a solution.

3.2 Domain Theory and Specification in KIDS

When designing an algorithm to solve a problem, one of the the most difficult parts is the conceptualization of the problem itself. Furthermore, the algorithm for solving a problem depends highly on the way the problem is specified. Before a formal specification can be written, the relevant concepts, properties and relationships of the problem must be defined - do not count much on KIDS to formalize the domain theory for solving a problem. This is essentially an exercise that requires a full understanding of the domain and a good understanding of how KIDS operates.

A *domain theory* consists of a model of the domain, expressed in some formal language (e.g., functional language, first order logic), regarding inputs, outputs, operations, constraints and objectives. KIDS uses a functional specification programming language augmented with set-theoretic data types. The main type of constructors and their operations are based on those of the REFINE language. This language supports (a subset of) first-order logic, set theory, pattern matching, and transformation rules. Refine has its own compiler that generates CommonLisp code.

A *domain theory* in KIDS comprises of:

- imported theories
- new type definitions
- function specifications
- laws (axioms and theorems)

- rules of inference

In KIDS a *specification* is represented by a quadruple $F =< D, R, I, O >$, where D is the input type subject to the *input condition*, $I : D \rightarrow boolean$, that defines legal inputs. The output type is R, and the *output condition*, $O : D \text{ X } R \rightarrow boolean$, defines the notion of feasible solution. If $O(x, z)$, then z is a feasible solution with respect to input x. In KIDS the specification of a program follows the format [Smith 87]:

function $F(x : D) : set(R)$

where $I(x)$

returns $\{z|O(x, z)\} = Body(x)$

This problem specification for F returns the *set of all* the values z of type R that satisfy the output condition O.

A *specification* for program F is *consistent* if for all possible input satisfying the input condition, the body produces all the feasible solutions z i.e., [Smith 87],

$$\forall(x : D)(I(x) => \{z|O(x, z)\}).$$

Even though the current implementation of KIDS only provides elementary support to the design of domain theories and specification of problems, there are some handy features in KIDS that alleviate those phases of program development. In particular, KIDS provides some support for the the derivation of distributive laws as well as it facilitates a hierarchical structure of programming through the importation of other theories into a particular domain theory.

3.3 Domain Theory and Specification for the K-MC PROBLEM

In order to encode the K-MC PROBLEM in KIDS we adopted the conventions displayed in table 1. Moves are represented by the set $\{1 \ldots 10\}$. As an example, the move of one missionary and one cannibal from the left bank to the right bank is represented by 1. The effect of moving one missionary and one cannibal from the left bank to the right bank in terms of the number of cannibals on the left bank is -1 and in terms of the number of missionaries on the left bank is also -1.

Once we have defined the conventions to represent the different moves $\{1 \ldots 10\}$, the specification of a domain theory for the K-MC PROBLEM in KIDS consists of translating the following statements into KIDS language:

Given k (missionaries and cannibals and a 2-person boat on the left bank), find all the *sequences of moves* such that:

Type of Move	Convention	Variation of Number of Cannibals on Left Bank (mapcl)	Variation of Number of Missionaries on Left Bank (mapml)
1 missionary and 1 cannibal from left bank to right bank	1	-1	-1
2 missionaries from left bank to right bank	2	0	-2
2 cannibals from left bank to right bank	3	-2	0
1 missionary from left bank to right bank	4	0	-1
1 cannibal from left bank to right bank	5	-1	0
1 missionary and 1 cannibal from right bank to left bank	6	1	1
2 missionaries from right bank to left bank	7	0	2
2 cannibals from right bank to left bank	8	2	0
1 missionary from right bank to left bank	9	0	1
1 cannibal from right bank to left bank	10	1	0

Table 1: The moves of the missionaries and cannibals problem

1 All the moves belong to the set of valid moves (according to our convention $\{1 \ldots 10\}$);

2 Moves from the left bank to the right bank alternate with moves from the right bank to the left bank;[3]

3 No more cannibals (missionaries) than the number of cannibals (missionaries) on a bank can be moved from that bank to the other bank;

4 The number of cannibals cannot exceed the number of missionaries on either bank;

5 The same state cannot be visited more than once;[4]

6 After the last move, all the cannibals, all the missionaries, and the boat have to be on the right bank;

Note the declarative nature of the statements that define the problem. In other words, the above statements simply state what the solution should follow in order to be a valid solution. We do not state how to actually generate the solution. That is provided by the search tactic.

The translation of the above statements into KIDS is the following top level function:

```
function MC-TOP (K: integer | K >= 1)
  returns
    (MC-ALL: set(seq(integer))
       | MC-ALL
           = {MC | (MC: seq(integer))
                 range(MC) subset {1 .. 10} and ALTERNATE-MOVE(MC)
                   and POSSIBLE-MOVE
                       (VISITED-STATES(MC, <K, K, 1>), K)
                   and NO-CANNIBALISM
                       (VISITED-STATES(MC, <K, K, 1>), K)
                   and NO-MOVES-BACK(VISITED-STATES(MC, <K, K, 1>))
                   and EVERYBODY-MADE-IT(VISITED-STATES(MC, <K, K, 1>))})
```

MC-ALL is the set of all solutions for the K-MC PROBLEM. Each solution MC is a sequence of integers within the range $\{1 \ldots 10\}$ and satisfying the constraints embodied by the functions: ALTERNATE-MOVE, POSSIBLE-MOVE, NO-CANNIBALISM, NO-MOVES-BACK, AND EVERYBODY-MADE-IT

The function ALTERNATE-MOVE guarantees that moves from the left bank to the right bank are alternated with moves from the right bank to the left bank. It corresponds to constraint 2 of the set of constraints stated above. It is defined as follows:

[3]Note that in our conceptualization we did not represent the boat explicitly. This condition, together with condition 1, implicitly considers the boat, guaranteeing that the moves alternate from one bank to the other, starting on the left bank.

[4]Without this condition there would be infinite solutions.

```
function ALTERNATE-MOVE (MC: seq(integer)): boolean
  = fa (I: integer)
      (I in [1 .. size(MC)]
        => (I mod 2 ~= 0 and MC(I) < 6)
             or (I mod 2 = 0 and MC(I) > 5))
```

Basically, it states that the even positions of the sequence have to be less than 6 and the odd positions of the sequence have to be greater than 5. Notice that according to our convention moves from the left bank to the right bank correspond to the range $\{1 \ldots 5\}$, while moves from the right bank to the left bank correspond to the range $\{6 \ldots 10\}$.

In order to implement the other constraints, we defined the function VISITED-STATES that returns the sequence of states the system goes through. The state is defined by the tuple $< clb, mlb, boat >$, where clb is the number of cannibals on the left bank, mlb is the number of missionaries on the left bank, and boat is the bank where the boat is.[5] Notice that since we know that we have k missionaries and k cannibals, we know the number of missionaries on the right bank, $mrb = k - mlb$, and the number of cannibals on the right bank, $crb = k - clb$.

The function POSSIBLE-MOVE corresponds to constraint 3 of the set of constraints listed above. It specifies that at each state the system goes through $clb \leq 0, crb \leq 0, mlb \leq 0, mrb \leq 0$. The function POSSIBLE-MOVE is defined as follows:

```
function POSSIBLE-MOVE
  (S: seq(tuple(integer, integer, integer)), K): boolean
  =  fa (I:integer)
       (I in domain(S)
            => STATE(I).1 >= 0 and STATE(I).2 >= 0
              and K - STATE(I).1 >= 0 and (K - STATE(I).2) >= 0)
```

The function NO-CANNIBALISM guarantees that cannibals never outnumber missionaries, on either bank (constraint 4). It is defined as follows:

```
function NO-CANNIBALISM
  (S: seq(tuple(integer, integer, integer)), K: integer)
  : boolean
  =  fa (I:integer)
       (I in domain(S)
```

[5] According to our convention, 1 corresponds to the left bank and -1 to the right bank.

```
=> (STATE(I).2 = 0 or STATE(I).2 >= STATE(I).1)
  and (K - STATE(I).2 = 0
       or K - STATE(I).2 >= K - STATE(I).1))
```

The function NO-MOVES-BACK guarantees that the same state is not re-visited (constraint 5), and it is defined as follows:

```
function NO-MOVES-BACK
  (VIS-STATES: seq(tuple(integer, integer, integer))): boolean
  = fa (I: integer, J: integer)
      (I in [1 .. size(VIS-STATES)]
         and J in [1 .. size(VIS-STATES)] and I ~= J
         => VIS-STATES(I) ~= VIS-STATES(J))
```

Finally, the function EVERYBODY-MADE-IT guarantees that after the last move, all the cannibals, missionaries, and the boat have to be on the right bank, which is equivalent to saying that the number of cannibals on the left bank is 0, the number of missionaries on the left bank is 0 and the boat is on the right bank.

```
function EVERYBODY-MADE-IT
  (VIS-STATES: seq(tuple(integer, integer, integer))): boolean
  = (last(VIS-STATES) = <0, 0, -1>)
```

3.3.1 Distributive Laws

Laws are assertions that define axioms or theorems, i.e., statements that are always true.[6] In KIDS distributive and monotonic laws are commonly used. The idea is to provide information on alternative ways of defining predicates, exactly in the same way one would write a law about how to distribute multiplication over addition. Additionally, laws also specify special cases, for instance when dealing with base cases (e.g., empty sequences).

KIDS provides some rudimentary support for the the derivation of distributive laws. However, in general the laws derived automatically by KIDS are very poor and not simple enough for the directed inference mechanism to be able to use them. For instance, using the laws derived automatically by KIDS for the MC-PROBLEM the directed inference mechanism could not derive any filter (see section 3.4.1). A useful heuristic in writing laws about the domain is that they should be simple, normally expressed in terms of the main function and perhaps another

[6]The difference between an assertion and a function is the following. A function always returns a value. An assertion is simply a true statement - e.g., $(A + B) * C = (A * C) + (B * C)$, or (A and $B \rightarrow A$)

function to handle cross-products. The distributive laws for the function ALTERNATE-MOVE are listed below. Notice the cross function, SHIFT-ALTERNATE-MOVE, defined as follows:

```
function SHIFT-ALTERNATE-MOVE
  (S: seq(integer), OFFSET: integer): boolean
  = fa (I: integer)
      (I in domain(S)
         => (I + OFFSET) mod 2 ~= 0 and S(I) < 6
              or (I + OFFSET) mod 2 = 0 and S(I) > 5)



assert DISTRIBUTE-ALTERNATE-MOVE-OVER-EMPTY-SEQ
  fa () ALTERNATE-MOVE([]) = true

assert DISTRIBUTE-ALTERNATE-MOVE-OVER-SEQUENCE-CONCATENATE
  fa (S1, S2)
    ALTERNATE-MOVE(S1 ++ S2)
      = (ALTERNATE-MOVE(S1) and SHIFT-ALTERNATE-MOVE(S2, size(S1)))

assert
  DISTRIBUTE-ALTERNATE-MOVE-OVER-PREPEND-ELEMENT-TO-SEQUENCE
  fa (A, S)
    ALTERNATE-MOVE(prepend(S, A))
      = (ALTERNATE-MOVE([A]) and SHIFT-ALTERNATE-MOVE(S, 1))

assert
  DISTRIBUTE-ALTERNATE-MOVE-OVER-APPEND-ELEMENT-TO-SEQUENCE
  fa (A, S)
    ALTERNATE-MOVE(append(S, A))
      = (ALTERNATE-MOVE(S) and SHIFT-ALTERNATE-MOVE([A], size(S)))
```

The complete domain theory for the MC PROBLEM is listed in the appendix (section 6.1)

3.4 Algorithm Design in KIDS

After defining a domain theory, the next step is to develop a high-level algorithm for enumerating solutions. The search tactic that we used for the MC-PROBLEM was *global search* [Smith 87].

This tactic designs a backtrack algorithm, a refinement of generate-and-test. The tactic is implemented by finding a space containing all the solutions to the problem that can be

divided into nested subspaces. The global search algorithm starts with an initial set that contains all the solutions to the given problem instance, the algorithm repeatedly extracts solutions, splits sets, and eliminates sets using filters until no sets remain to be split. The process can be described as a tree search in which a node represents a set of candidates, and an arc represents the split relationship between a set and a subset. The principal operations are to extract candidate solutions from a set and to split a set into subsets. In this model of global search, solutions can be extracted from all nodes of the tree, not just the leaves. The derivation of efficient *filters* that prune subspaces that do not contain any feasible solution is a complementary operation in the derivation of the global search tactic. The pruning mechanism is derived by forward inference from the feasibility condition.

In order to use global search as the search paradigm for a given *specification*, the problem specification represented by a quadruple $F =< D, R, I, O >$ is extended with the abstract data type that represents sets of solutions, $\hat{R}$. Formally, a *global search theory* (*gs–theory*) is presented as follows [Smith 87]:

Sorts

D	*input domain*
R	*output domain*
$\hat{R}$	*subspace descriptors*

Operations

$I : D \rightarrow boolean$	*input condition*
$O : D \times R \rightarrow boolean$	*input/output condition*
$\hat{I} : D \times \hat{R} \rightarrow boolean$	*subspace descriptors condition*
$\hat{r}_0 : D \rightarrow \hat{R}$	*initial space*
$Satisfies : R \times \hat{R} \rightarrow boolean$	*denotation of descriptors*
$Split : D \times \hat{R} \times \hat{R} \rightarrow boolean$	*split relation*
$Extract : R \times \hat{R} \rightarrow boolean$	*extractor of solutions from spaces*

Axioms

GS0. $I(x) \implies \hat{I}(x, \hat{r}_0(x))$

GS1. $I(x) \wedge \hat{I}(x, \hat{r}) \wedge Split(x, \hat{r}, \hat{s}) \implies \hat{I}(x, \hat{s})$

GS2. $I(x) \wedge O(x, z) \implies Satisfies(z, \hat{r}_0(x))$

GS3. $I(x) \wedge \hat{I}(x, \hat{r}) \implies (Satisfies(z, \hat{r}) \iff \exists(\hat{s})\ (\ Split^*(x, \hat{r}, \hat{s}) \wedge Extract(z, \hat{s})))$

The sorts and operators that extend the problem specification in the GS-theory are denoted with hats: $\hat{R}$ is the space descriptor that represents sets of solutions (elements of $\hat{R}$ are denoted in lowercase); $\hat{I}$ is an operator on $\hat{R}$ that defines legal sets of solutions; $Satisfies(z, \hat{r})$ means that the solution z is in the set denoted by descriptor $\hat{r}$; $Split(x, \hat{r}, \hat{s})$ means that $\hat{s}$ is a subspace of $\hat{r}$ with respect to input x; and $Extract(z, \hat{r})$ means that z is directly extractable from $\hat{r}$. Axiom GS0 states that the initial descriptor $\hat{r}_0(x)$ is a legal descriptor. Axiom GS1 states that legal space descriptors split into legal descriptors. Axiom GS2 states that all feasible solutions are contained in the initial space descriptor, $\hat{r}_0$. Axiom GS3 states that an

output object z is in the set denoted by $\hat{r}$ if and only if z can be extracted after applying *Split* to $\hat{r}$ a finite number of times, which is denoted by the *.

A gs-theory can be represented as a tuple

$$\mathcal{G} = \langle \mathcal{B}, \hat{R}, \hat{I}, \hat{r}_0, Satisfies, Split, Extract \rangle$$

where $\mathcal{B} = \langle D, R, I, O \rangle$.

For more details about gs-theory and its *well-foundedness* see [Smith 87].

An example of a global search theory follows:

$$\begin{array}{rcl}
F & \mapsto & gs - sequences - over - finite - set \\
D & \mapsto & set(\alpha) \\
I & \mapsto & \lambda(S)\ true \\
R & \mapsto & seq(\alpha) \\
O & \mapsto & \lambda(S,q)\ range(q) \subseteq S \\
\hat{R} & \mapsto & seq(\alpha) \\
\hat{I} & \mapsto & \lambda(S,V)\ range(V) \subseteq S \\
Satisfies & \mapsto & \lambda(q,V)\ extends(q,V) \\
\hat{r}_0 & \mapsto & [\,] \\
Split & \mapsto & \lambda(S,V,V')\ \exists(i)(i \in S \Longrightarrow V' = append(V,i)) \\
Extract & \mapsto & \lambda(q,V)\ q = V
\end{array}$$

This global search theory enumerates sequences over a finite set of generic type α, S - the output condition imposes that the output sequences q are subsets of S. V is the subspace descriptor - the initial subspace descriptor is the [ ] and legal space descriptors are subsets of S. *Splitting* corresponds to appending an element from S into V. The output sequence can be extracted from the subsapce whose descriptor is V.

The following theorem defines a recursive program specification to implement an abstract global search theory.

Theorem 3.1 *If $\mathcal{G}_F$ is a well-founded gs-theory then the following multilinear recursive program specification is consistent.*

function $F(x : D) : set(R)$
  where $I(x)$
  returns $\{z \mid O(x,z)\}$
  $= F\_gs(x, \hat{r}_0(x))$

function $F\_gs(x : D, \hat{r} : \hat{R}) : set(R)$
  where $I(x) \wedge \hat{I}(x, \hat{r})$
  returns $\{z \mid Satisfies(z, \hat{r}) \wedge O(x,z)\}$
  $= \{z \mid Extract(z, \hat{r}) \wedge O(x,z)\}$
  $\cup\ reduce(\cup, \{ F\_gs(x, \hat{s}) \mid Split(x, \hat{r}, \hat{s})\})$.

The proof of the theorem can be found in [Smith 87].

Once we have a program for a given global search theory, it is important to define how we can use that program for a given problem specification. The following theorem defines how to construct a global search program for a general problem $\mathcal{B}_F = \langle D_F, R_F, I_F, O_F \rangle$ from a given global search theory, $\mathcal{B}_G = \langle D_G, R_G, I_G, O_G \rangle$, and assuming that there is a translation $t(x)$ that specializes $\mathcal{B}_F$ to $\mathcal{B}_G$, i.e.:

Definition. *$\mathcal{B}_F$ specializes $\mathcal{B}_G$* if $R_F = R_G$ and

$$\forall(x : D_F)\exists(y : D_G)\forall(z : R_F)((I_F(x) \implies I_F G(y)) \wedge (O_F(x,z) \implies O_G(y,z))). \quad (1)$$

Definition. *$\mathcal{B}_F$ specializes* to $\mathcal{B}_G$ with translation $t(x)$ if

$$\forall(x : D_F)\forall(z : R_F)((I_F(x) \implies I_F G(t(x))) \wedge (O_F(x,z) \implies O_G(t(x),z))).$$

Theorem 3.2 *Let $\mathcal{G}_G = \langle \mathcal{B}_G, \hat{R}, \hat{I}, \hat{r}_0, \text{Satisfies}, Split, Extract \rangle$ be a global search theory, and let $\mathcal{B}_F = \langle D_F, R_F, I_F, O_F \rangle$ be a problem theory that specializes $\mathcal{B}_G = \langle D_G, R_G, I_G, O_G \rangle$ with translation $t(x)$, then the structure $\mathcal{G}_F = \langle \mathcal{B}_F, \hat{R}, \lambda(x,\hat{r})\hat{I}(t(x),\hat{r}), \text{Satisfies}, \lambda(x)\hat{r}_0(t(x)),$ $\lambda(x,\hat{r},\hat{s})$ $Split(t(x),\hat{r},\hat{s}), Extract \rangle$ is a global search theory.*

The proof of the theorem can be found in [Smith 87].

3.4.1 Filters

In order to reduce the search space, whenever possible KIDS automatically derives necessary *filters*, i.e., predicates that are used to prune off branches of the search tree that do not contain feasible solutions.

The rationale underlying the derivation of *necessary filters* is that a feasible solution $(z : R)$ in a given space $\hat{r}$ verifies the following condition:

$$(\ Satisfies(z,\hat{r}) \wedge O(x,z)\ ) \quad (2)$$

Therefore, necessary conditions of this condition can be used as filters, $\psi(x,\hat{r})$;, i.e.,:

necessary feasibility filters, where

$$\exists(z : R)\ (\ Satisfies(z,\hat{r}) \wedge O(x,y)\ ) \implies \psi(x,\hat{r}); \quad (3)$$

In other words, if $\neg\psi(x,\hat{r})$ then by the contrapositive of 3, there are no feasible solutions in the space denoted by $\hat{r}$).

Necessary feasibility filters, denoted by Φ, are defined by the condition

$$\forall(x : D)\, \forall(\hat{r} : \hat{R})\, \forall(z : R)\, (\, I(x) \wedge \hat{I}(x, \hat{r}) \Longrightarrow (\mathit{Satisfies}(z, \hat{r}) \wedge O(x, z) \Longrightarrow \Phi(x, \hat{r}))). \tag{4}$$

A global search program incorporating necessary filters is defined by the following proposition [Smith 87]:

Proposition 3.1 *If $\mathcal{G}_F$ is a well-founded gs-theory and Φ is a necessary feasibility filter, then the following program specification is consistent.*

function $F(x : D) : set(R)$
  where $I(x)$
  returns $\{z \mid O(x, z)\}$
  $=$ **if** $\Phi(x, \hat{r}_0(x))$**then** $F\_gs(x, \hat{r}_0(x))$**else** $\{\}$

function $F\_gs(x : D, \hat{r} : \hat{R}) : set(R)$
  where $I(x) \wedge \hat{I}(x, \hat{r}) \wedge \Phi(x, \hat{r})$
  returns $\{z \mid \text{Satisfies}(z, \hat{r}) \wedge O(x, z)\}$
  $= \{z \mid \mathit{Extract}(z, \hat{r}) \wedge O(x, z)\}$
  $\cup\, \mathit{reduce}(\cup, \{ F\_gs(x, \hat{s}) \mid \mathit{Split}(x, \hat{r}, \hat{s}) \wedge \Phi(x, \hat{s})\}).$

3.5 A First Version of a Program for the K-MC PROBLEM

In practical terms, the creation of a program in KIDS using a given global search algorithm involves three steps:

1. The user selects a global search theory from a library that solves the problem of enumerating the output type for the given problem - currently KIDS library includes global search theories for a number of problem domains such as enumerating sets, sequences, maps and integers.
2. KIDS automatically finds a substitution θ for which the problem to be solved completely reduces to the problem of the same type for which KIDS has a global search theory;
3. KIDS automatically derives a necessary filter - filters are predicates that are used to eliminate subspaces of the search space that do not contain any feasible solution from further consideration during search

1 Selection of a Search Tactic

In the case of the MC-PROBLEM, we selected the global search theory for enumerating sequences, *gs-sequences-over-finite-set*, since the output type of the problem is sequence.

2 Reduction to the Selected Search Tactic

The MC-PROBLEM complete reduces to the gs-theory *gs-sequences-over-finite-set.* Instantiating the definition of specialization (1) (see page 14), we obtain the following expression:

$\forall(k : integer)\exists(S : set(\alpha))\forall(MC : seq(integer))$

$(range(MC) \subseteq \{1..10\}$
$\wedge$ ALTERNATE-MOVE(MC)
$\wedge$ POSSIBLE-MOVE(VISITED-STATES(MC, <K, K, 1>), K)
$\wedge$ NO-CANNIBALISM(VISITED-STATES(MC, <K, K, 1>), K)
$\wedge$ NO-MOVES-BACK(VISITED-STATES(MC, <K, K, 1>))
$\wedge$ EVERYBODY-MADE-IT(VISITED-STATES(MC, <K, K,1>))
$\Longrightarrow$
$(range(MC) \subseteq S))$

The proof is straightforward and results in the translation:

$$\{S \mapsto \{1..10\}\}.$$

KIDS automatically finds the translation for which the problem to be solved completely reduces to the problem of the same type selected by the user, if such a translation exists. An excerpt of the inference performed by Rainbow[7] to find the translation for the K-MC PROBLEM is listed in the appendix (section 6.2).

3 Filters

The derivation of filters is performed automatically by KIDS using the directed inference mechanism to derive necessary conditions on the existence of solutions for a subspace descriptor V. We came to the conclusion that, in order for Rainbow to derive "good" filters, it is important to have distributive laws about all the predicates that define the output condition.

An excerpt of the inference performed by Rainbow when deriving filters for the MC-PROBLEM is listed in the appendix (section 6.3). Notice the usage of the distributive laws by Rainbow in order to distribute the predicates that define the output condition over the output sequence MC, considering that the output sequence MC can be written as a function of the space descriptor V as: MC = concat(V, Y-197).

From all the derived terms, KIDS proposes as filters the ones that are function of the input variable k and the space descriptor V. Several of the filters proposed by KIDS are redundant

[7]Rainbow is the name of the theorem prover in KIDS.

or do not have any pruning impact - a good example of that is the filter "true". From the filters proposed by KIDS, we selected the significant ones to be incorporated into the global search algorithm:

```
NO-CANNIBALISM(VISITED-STATES(V, <K, K, 1>), K)
  & POSSIBLE-MOVE(VISITED-STATES(V, <K, K, 1>), K)
  & NO-MOVES-BACK(VISITED-STATES(V, <K, K, 1>))
  & ALTERNATE-MOVE(V)

New global search theory incorporating the necessary
filter:(operator-specification MC-TOP number-of-solutions ALL
  input-types integer output-types seq(integer) input-vars K
  output-vars MC input-condition K >= 1
  output-condition
    range(MC) subset {1 .. 10} & ALTERNATE-MOVE(MC)
      & POSSIBLE-MOVE(VISITED-STATES(MC, <K, K, 1>), K)
      & NO-CANNIBALISM(VISITED-STATES(MC, <K, K, 1>), K)
      & NO-MOVES-BACK(VISITED-STATES(MC, <K, K, 1>))
      & EVERYBODY-MADE-IT(VISITED-STATES(MC, <K, K, 1>)))
```

As a result of the previous steps a well-structured but inefficient program is automatically created by KIDS, using the global search tactic and incorporating as filters:

- ALTERNATE-MOVE(V)
- NO-MOVES-BACK(VISITED-STATES(V, <K, K, 1>))
- POSSIBLE-MOVE(VISITED-STATES(V, <K, K, 1>), K)
- NO-CANNIBALISM(VISITED-STATES(V, <K, K, 1>), K)

```
function MC-TOP-AUX
  (K: integer, V: seq(integer)
     | K >= 1 & range(V) subset {1 .. 10} & ALTERNATE-MOVE(V)
         & NO-MOVES-BACK(VISITED-STATES(V, <K, K, 1>))
         & POSSIBLE-MOVE(VISITED-STATES(V, <K, K, 1>), K)
         & NO-CANNIBALISM(VISITED-STATES(V, <K, K, 1>), K))
  returns
    (MC-TOP-SET: set(seq(integer))
       | MC-TOP-SET
           = {MC | (MC: seq(integer))
                EXTENDS(MC, V)
                  & EVERYBODY-MADE-IT(VISITED-STATES(MC, <K, K, 1>))
```

```
                & NO-MOVES-BACK(VISITED-STATES(MC, <K, K, 1>))
                & NO-CANNIBALISM(VISITED-STATES(MC, <K, K, 1>), K)
                & POSSIBLE-MOVE(VISITED-STATES(MC, <K, K, 1>), K)
                & ALTERNATE-MOVE(MC) & range(MC) subset {1 .. 10}})
  = {MC | (MC: seq(integer))
      EVERYBODY-MADE-IT(VISITED-STATES(MC, <K, K, 1>))
        & NO-MOVES-BACK(VISITED-STATES(MC, <K, K, 1>))
        & NO-CANNIBALISM(VISITED-STATES(MC, <K, K, 1>), K)
        & POSSIBLE-MOVE(VISITED-STATES(MC, <K, K, 1>), K)
        & ALTERNATE-MOVE(MC) & range(MC) subset {1 .. 10} & MC = V}
     union! reduce
              (UNION!,
               {MC-TOP-AUX(K, NEW-V) | (NEW-V: seq(integer))
                  ALTERNATE-MOVE(NEW-V)
                    & NO-MOVES-BACK(VISITED-STATES(NEW-V, <K, K, 1>))
                    & POSSIBLE-MOVE(VISITED-STATES(NEW-V, <K, K, 1>), K)
                    & NO-CANNIBALISM(VISITED-STATES(NEW-V, <K, K, 1>), K)
                    & ex (I: integer)
                        (NEW-V = append(V, I) & I in {1 .. 10})})

function MC-TOP (K: integer | K >= 1)
  returns
    (MC-TOP-SET: set(seq(integer))
       | MC-TOP-SET
           = {MC | (MC: seq(integer))
                range(MC) subset {1 .. 10} & ALTERNATE-MOVE(MC)
                  & POSSIBLE-MOVE(VISITED-STATES(MC, <K, K, 1>), K)
                  & NO-CANNIBALISM(VISITED-STATES(MC, <K, K, 1>), K)
                  & NO-MOVES-BACK(VISITED-STATES(MC, <K, K, 1>))
                  & EVERYBODY-MADE-IT(VISITED-STATES(MC, <K, K, 1>))})
  = if NO-CANNIBALISM(VISITED-STATES([], <K, K, 1>), K)
       & POSSIBLE-MOVE(VISITED-STATES([], <K, K, 1>), K)
       & NO-MOVES-BACK(VISITED-STATES([], <K, K, 1>))
       & ALTERNATE-MOVE([])
    then MC-TOP-AUX(K, [])
    else {}
```

3.6 Program Optimization in KIDS

In general, the code produced by KIDS incorporating a search tactic is well structured but very inefficient, with several opportunities for optimization. KIDS provides several tools for

program optimization. In the following sections we illustrate some of such optimization tools.

KIDS generates the latex code for reporting the derivation process. In the following examples of optimizations, the latex code describing the "before" and "after" was actually produced by KIDS. KIDS keeps track of the "history" of a given derivation, i.e., the sequence of transformations applied to the code. At any point of the derivation, the user can restore the program as it was at an earlier stage. The user can also replay a derivation.

3.6.1 Context Independent Simplifier

3.6.1.1 Type 1 It consists of a set of equations treated as left-to-right rewrite rules that are fired exhaustively until none apply. An examples of a rewrite rule:

- *if true then P else Q = P*

Distributive laws are also treated as rewrite rules.

3.6.1.2 Type 2 Another form of *Context Independent Simplifier* is done by replacing all the occurrences of a local variable that is defined by an equality.:

$\{C(x)|x = e \ \& \ P(x)\} = C(e)|P(e)$

In appendix (6.3.2) an example of a context-independent simplification performed by KIDS is listed. It mainly uses the distributive laws regarding the different predicates that define the output condition for the base case [ ], where they simplify to *true*.

3.6.2 Context Dependent Simplifier

This form of simplification is designed to simplify a given expression with respect to its context. The way it works is gathering all the predicates that hold in the context of the expression by walking up the abstract syntax tree. The expression is then simplified with respect to the set of assumptions that hold in the context [Smith 91].

In appendix 6.4 an exampleof a context dependent simplification is shown.

3.6.3 Finite Differencing

Roughly, the idea behind finite differencing is to perform computations incrementally rather than recomputing them from scratch all the time. As an example let's assume that inside $f(x)$ there is a function $g(x)$ and that x changes in a regular way. In this case, it might be worthwhile to create a new variable, equal to $g(x)$, whose value is maintained and which

allows for incremental computation. Finite differencing can be decomposed into two more basic operations: abstraction and simplification [Smith 90, Smith 91].

- Abstraction of a function f with respect to expression $g(x)$ adds a new parameter c to f's parameter list (now $f(x, c)$) and adds $c = g(x)$ as a new input invariant to f. Any call to f, whether a recursive call within f or an external call, must now be changed to supply the appropriate new argument that satisfies the invariant - $f(x)$ is changed to $f(x, g(x))$.
- Simplification - context Independent simplification is applied to the new argument in all external calls.

In this process all occurrences of $g(x)$ are replaced by c. Often, distributive laws apply to $g(U(x))$ yielding an expression of the form $U'(g(x))$ and so $U'(c)$. The real benefit in the optimization comes from the last step, because this is where the new value of the expression $g(U(x))$ is computed in terms of the old value of $g(x)$.

The process of finite differencing involves:

- the selection by the user of a function to be finite differenced - since the space descriptor V grows incrementally, a good hint is to finite difference functions that have V as argument. Normally one discovers interesting meaningful abstractions by finite differencing, which is reflected in the name of the variable that replaces the function.
- the replacement of all the occurrences of the function by the corresponding variable - this task is performed automatically by KIDS
- simplification (manually driven)

As an example, the function VISITED-STATES(V, <K,K,1>) is finite differenced into the variable VS, which represents the set of states visited. The function SIZE(V) is finite differenced into the variable NUMBER-MOVES. Note that V is the space descriptor and it represents the sequence of moves so far. Therefore, it makes sense to call the variable that corresponds to SIZE(V) as NUMBER-MOVES.

The "before' and "after" code regarding the finite differencing of VISITED-STATES(V, <K,K,1>) is listed in the appendix (see section 6.5).

3.7 Final Program for the MC PROBLEM

A summary of the derivation steps for a program for the MC PROBLEM is listed below. This summary was automatically generated by KIDS in latex code.

The optimized code is considerably faster than the initial code - the initial code finds the solutions for 100 problems (K-MC PROBLEM, $k = 1 \ldots 100$) in 3.67 hours, while the optimized code only takes 3.75 minutes, real time.

```
SUMMARY
 1. Focus Initialize MC-TOP
 2. Tactic Global Search on MC-TOP
 3. Simplify, context-independent-fast: if ## & ## & ## & ## then MC-TOP-aux...
 4. Simplify, context-independent-fast: {MC | (##) ##} union! reduce(UNION!...
 5. Simplify, context-dependent, forward-0, backward-4:  ## subset ## &...
 6. Simplify, context-dependent, forward-0, backward-4:  ##(...) & ##(....
 7. Simplify, context-independent-fast: K >= 1
 8. Simplify, context-dependent, forward-0, backward-4:  0 < K
 9. Simplify, context-independent-fast: if true then MC-TOP-AUX(##, ##) ...
10. FD (general-purpose) VS = VISITED-STATES(V, <K, K, 1>)
11. Unfold ALL-BUT-1ST-VISITED-STATES([##], last...
12. Unfold VISITED-STATES([I], last(VS)) using rewrite rule UNFOLD-FUNCTION
13. Simplify, context-independent-fast: rest(if empty(##) then [##] else [#...
14. Abstract ALL-BUT-1ST-VISITED-STATES([I], last(VS)) into
CURRENT-STATE in {MC-TOP-AUX(##, ##, ##) ...
15. Unfold ALL-BUT-1ST-VISITED-STATES([##], last...
16. Unfold VISITED-STATES([I], last(VS)) using rewrite rule UNFOLD-FUNCTION
17. Simplify, context-independent-fast: rest(if empty(##) then [##] else [#...
18. FD (general-purpose) NUMBER-NODES = size(V)
19. FD (general-purpose) CURRENT-STATE = last(VS)
20. Simplify, context-independent-fast: last(VS ++ [<##, ##, ##>])
21. Unfold EVERYBODY-MADE-IT(VS) using rewrite rule UNFOLD-FUNCTION
22. Simplify, context-dependent, forward-0, backward-4:  last(VS)
23. Unfold SHIFT-ALTERNATE-MOVE([I], V-SIZE)
24. Simplify, context-independent-fast: fa (I: integer) (I in domain(##) =>...
25. Abstract MAPCL(I) into MAPCL-I in {MC-TOP-AUX(##, ##, .....
26. Abstract MAPML(I) into MAPML-I in {MC-TOP-AUX(##, ##, .....
27. Refine compile into Lisp: MC-TOP-AUX, MC-TOP
```

The final version of the program for the K-MC PROBLEM that we derived using the sequence of transformations listed above is listed in the appendix (see section 6.6).

3.8 Complexity of the MC PROBLEM

Figure 1 illustrates the complete global search tree for the program for the K-MC PROBLEM derived with KIDS ($k = 1$). Each node of the tree represents the sequence of moves so far.

At the root of the tree, the empty sequence, represents no moves. Notice how the branches of the tree are pruned off by the filters. For instance, the branches starting with nodes [2], [3], [6], [7], [8], [9], and [10] are pruned off at the first level of the tree. In other words, for the case of 1-MC PROBLEM, where there is one cannibal and one missionary on the left bank in the initial state, the only moves that are valid (i.e., the ones that pass the filters) are: move 1 (move 1 missionary and 1 cannibal from the left to the right bank); move 4 (move 1 missionary from the left to the right bank); and move 5 (move 1 cannibal from the left bank to the right bank). All the other moves (2, 3, 6, 7, 8, 9, and 10) violate the filters - for instance, moves greater than 6 correspond to trying to move people from the left bank to the right bank when there are no people there. For the solutions starting with move 1, only the solutions starting with $[1, 9]$ are expanded to the third level and pruned off at this level. Solutions starting with move 4 and move 5 are pruned at the second level of the tree. Notice that in this model of global search, solutions can be extracted from all nodes of the tree, not only the leaves. Actually, for the 1-MC PROBLEM there is only one solution [1], i.e., the sequence with the single move 1 (one missionary and one cannibal from the left to the right bank).

The following table shows the number of expanded nodes for the program for the K-MC PROBLEM derived with KIDS, for different ks.

K	Number of Expanded Nodes
1	6
2	31
3	38
≥ 4	$4K + 4$

Table 2: The number of expanded nodes for different ks

Figure 2 depicts the global search tree for the K-MC PROBLEM, when $k \geq 4$. As k increases, the only additional moves that are feasible are: move 3, i.e., move two cannibals to the right bank, and move 10, i.e., move one cannibal back to the left bank. In other words, as k increases the only additional moves that are legal consist of transporting cannibals to the right bank by moving two cannibals to the right bank and having one cannibal bringing the boat back to the left bank. This way missionaries are safe, though they are stuck in the left bank. In other words, for $k \geq 4$, there is no solution for the K-MC PROBLEM, considering the generalization that we adopted. A better generalization of the 3-MC PROBLEM that would allow solutions for $k > 3$ corresponds to increasing the number of cannibals and missionaries as well as the size of the boat. The intuition is that we would no longer be dealing with a problem of linear complexity.

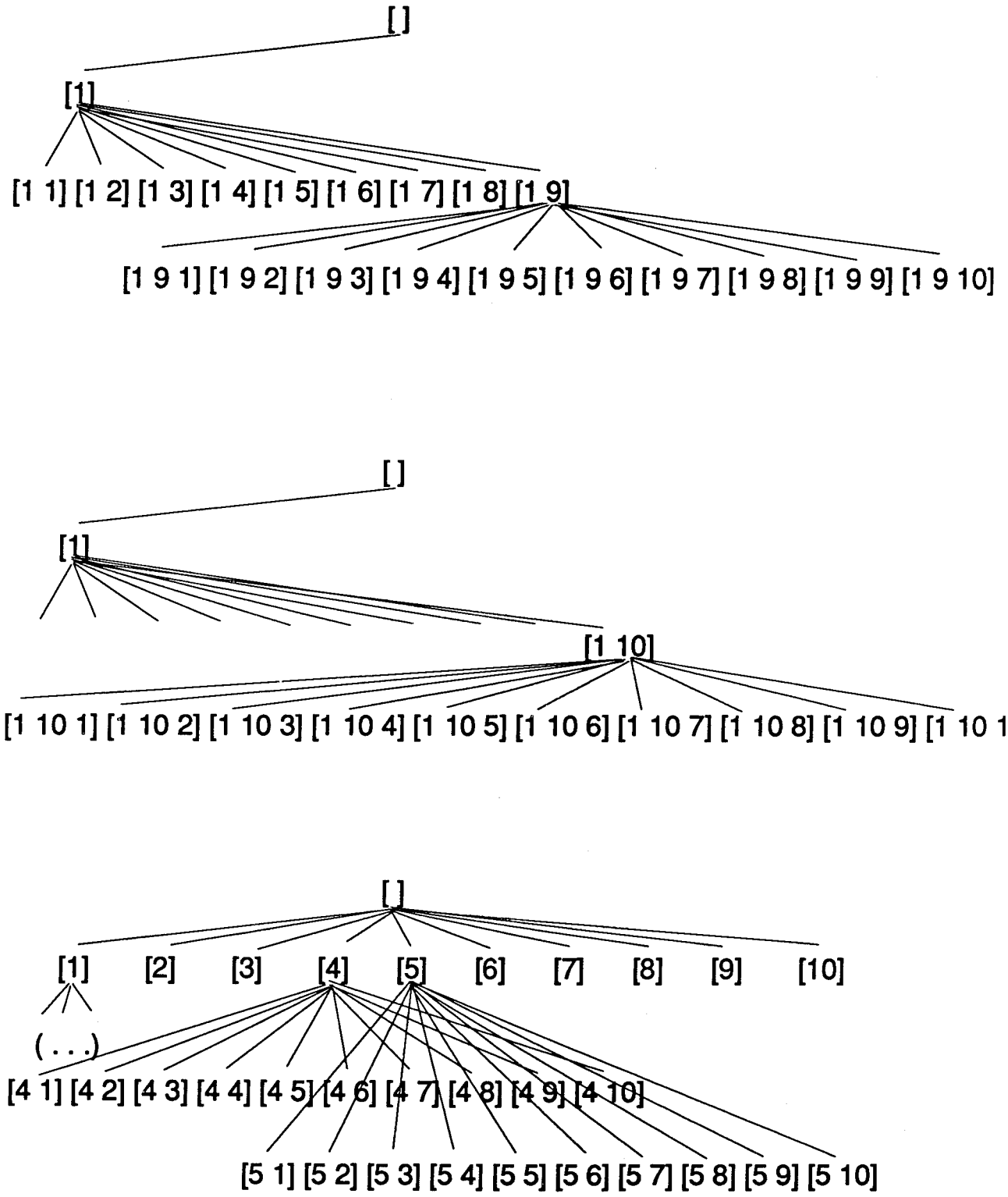


Figure 1: The Global Search Tree for the 1-MC PROBLEM

Figure 2: The Global Search Tree for the K-MC PROBLEM, $K \geq 4$ (expanded nodes)

4 AI Planners, KIDS, and Search

The Missionaries and Cannibals Problem (MC PROBLEM) is a classical AI planning problem. Our interest in this problem arose as the result of some difficulties that we experienced when trying to use the AI planners O-PLAN2 and SIPE-2 to solve it [Gomes 94] - SIPE-2 could not solve problems for $k > 45$ in less than 20 minutes real time. We derived a program in KIDS for the K-MC PROBLEM that clearly outperformed O-PLAN2 and SIPE-2. Why was that, when the formulation of the MC PROBLEM used in O-PLAN2 and SIPE-2 is identical to the formulation of it in KIDS? Furthermore, both the AI planners and KIDS are compiled in Lisp. The answer is the successive source-to-source transformations applied to the initial code in order to get an efficient implementation. The source-to-source transformations add a search startegy and control mechanism to the initial domain theory as well as they optimize the initial program derived by KIDS. As mentioned before, the initial code derived in KIDS finds the solutions for 100 problems (K-MC PROBLEM, $k = 1 \ldots 100$) in 3.67 hours, while the optimize code only takes 3.75 minutes, real time.

KIDS is not a planner. KIDS is an framework for the development of programs from formal specifications. KIDS is an interactive development system that provides tools for the design and development of correct and efficient programs from formal specifications. KIDS embodies a transformational approach to program development. The methodology of program development by program transformation consists of compiling, either manually or automatically, a formal specification into an efficient implementation by the repeated application of correctness-preserving, source-to-source transformations. Though the development methodologies for program transformation and for the design of efficient compilers share a lot in common, they constitute different approaches. The approach embodied in the design of compilers is mainly based on a local approach of syntax-directed translation to recursively decompose a program into elementary constructs that can be directly translated. The program transformation methodology emphasizes global analysis of high-level specifications through interaction with the user.

The most relevant feature of KIDS is its module for algorithm design and the approach that it embodies - the explicit recognition that different problems should be addressed with different search strategies and control mechanisms. For example, in the case of the K-MC PROBLEM, we adopted a global search tactic with backtracking and complete search - the program derived with KIDS finds all the solutions for the K-MC PROBLEM. But, as discussed in section 3.8, the K-MC PROBLEM has linear complexity for $k \geq 4$. Needless to say that most planning and scheduling problems, in particular real world problems, are intractable and therefore it is unrealistic to solve such problems with approaches that offer guarantees of soundness and completeness - a central concern in several AI planning approaches[8]. For such problems it might be more adequate to use local search strategies (e.g., simulated annealing, steepest ascent) or even greedy global search tactics with heuristic search. KIDS's philosophy, which is reinforced and expanded in its successor SPECWARE, is to provide a library of problem

[8]We are not claiming that the study of soundness and completeness is not relevant - we just consider that such approaches are not realistic when addressing intractable real world problems.

solving methods from Computer Science (e.g., divide and conquer, global search), Artificial Intelligence (e.g., constraint satisfaction paradigm), and Operations Research (e.g., Simplex, Integer Programming, etc). The idea is to provide a *classification approach* [Smith 92], i.e., a set of high level transformations embodying the different problem solving methods as well as high level transformation for refinement of a given paradigm into another, whenever applicable. As an example, a problem formulated as a constraint satisfaction problem might be refined successively into linear programming problem, integer problem and at the lower level an assignment problem.

5 Conclusions

In this paper we describe a transformational approach applied to a classical AI problem - the missionaries and cannibals problem. Our interest in this problem arose as the result of some difficulties that we experienced when trying to use the AI planners O-PLAN2 and SIPE-2 to solve it [Gomes 94] - SIPE-2 could not solve problems for $k > 45$ in less than 20 minutes real time. We derived a program in KIDS for the K-MC PROBLEM that clearly outperformed O-PLAN2 and SIPE-2. Other several small-scale problems have been solved using KIDS. The system has also been applied with success to larger, more realistic problems such as military transportation scheduling. We believe that a transformational approach would also be adequate to planning problems.

Acknowledgements

I would like to thank Doug Smith for his support with KIDS, comments on an earlier draft of this paper as well as for letting me use his latex macros. I also would like to thank Nancy Roberts, Karen Alguire, and John Crowter for their comments and their work with KIDS.

6 Appendix

6.1 The Domain Theory for the MC PROBLEM

```
THEORY MISSIONARIES

%-----------------------------------------------------------------------
THEORY-IMPORTS {}

%-----------------------------------------------------------------------
THEORY-TYPE-PARAMETERS {}

%-----------------------------------------------------------------------
THEORY-TYPES

constant MAPCL: map(integer, integer)
  = {| 1 -> -1, 2 -> 0, 3 -> -2,
       4 -> 0, 5 -> -1, 6 -> 1, 7 -> 0,
       8 -> 2, 9 -> 0, 10 -> 1
     |}

constant MAPML: map(integer, integer)
  = {| 1 -> -1, 2 -> -2, 3 -> 0,
       4 -> -1, 5 -> 0, 6 -> 1, 7 -> 2,
       8 -> 0, 9 -> 1, 10 -> 0
     |}

%-----------------------------------------------------------------------
THEORY-OPERATIONS

function MC-TOP (K: integer | K >= 1)
  returns
    (MC-ALL: set(seq(integer))
       | MC-ALL
           = {MC | (MC: seq(integer))
                range(MC) subset {1 .. 10} and ALTERNATE-MOVE(MC)
                  and POSSIBLE-MOVE
                      (VISITED-STATES(MC, <K, K, 1>), K)
                  and NO-CANNIBALISM
                      (VISITED-STATES(MC, <K, K, 1>), K)
                  and NO-MOVES-BACK(VISITED-STATES(MC, <K, K, 1>))
                  and EVERYBODY-MADE-IT(VISITED-STATES(MC, <K, K, 1>))})
```

```
function MC-SIZE (A: integer): seq(integer)
  = image
      (lambda (Y: integer) if empty(MC-TOP(Y)) then 0 else Y,
       [1 .. A])

function ALTERNATE-MOVE (MC: seq(integer)): boolean
  = fa (I: integer)
      (I in [1 .. size(MC)]
         => I mod 2 ~= 0 and MC(I) < 6
              or I mod 2 = 0 and MC(I) > 5)

function POSSIBLE-MOVE
  (S: seq(tuple(integer, integer, integer)), K): boolean
  = reduce
      ('AND,
       image
         (lambda (STATE: tuple(integer, integer, integer))
            STATE.1 >= 0 and STATE.2 >= 0
              and K - STATE.1 >= 0 and K - STATE.2 >= 0,
          S))

function SHIFT-ALTERNATE-MOVE
  (S: seq(integer), OFFSET: integer): boolean
  = fa (I: integer)
      (I in domain(S)
         => (I + OFFSET) mod 2 ~= 0 and S(I) < 6
              or (I + OFFSET) mod 2 = 0 and S(I) > 5)

function VISITED-STATES
  (MC: seq(integer), IS: tuple(integer, integer, integer))
  : seq(tuple(integer, integer, integer))
  = if empty(MC) then [IS]
    else
      [IS] ++ VISITED-STATES
                (rest(MC),
                 <IS.1 + MAPCL(first(MC)),
                  IS.2 + MAPML(first(MC)), IS.3 * -1>)

function ALL-BUT-1ST-VISITED-STATES
  (MC: seq(integer), IS: tuple(integer, integer, integer))
  : seq(tuple(integer, integer, integer))
  = rest(VISITED-STATES(MC, IS))

function NO-CANNIBALISM
  (S: seq(tuple(integer, integer, integer)), K: integer)
```

```
  : boolean
  = reduce
      ('AND,
       image
         (lambda (STATE: tuple(integer, integer, integer))
            (STATE.2 = 0 or STATE.2 >= STATE.1)
              and (K - STATE.2 = 0
                   or K - STATE.2 >= K - STATE.1),
          S))

function NO-MOVES-BACK
  (VIS-STATES: seq(tuple(integer, integer, integer))): boolean
  = fa (I: integer, J: integer)
      (I in [1 .. size(VIS-STATES)]
         and J in [1 .. size(VIS-STATES)] and I ~= J
         => VIS-STATES(I) ~= VIS-STATES(J))

function CROSS-NO-MOVES-BACK
  (R: seq(tuple(integer, integer, integer)),
   S: seq(tuple(integer, integer, integer)))
  : boolean
  = fa (I: integer, J: integer)
      (I in [1 .. size(R)]
         and J in [1 .. size(S)]
         => R(I) ~= S(J))
klk
function EVERYBODY-MADE-IT
  (VIS-STATES: seq(tuple(integer, integer, integer))): boolean
  = (last(VIS-STATES) = <0, 0, -1>)

%-----------------------------------------------------------------------
THEORY-LAWS

assert DISTRIBUTE-ALTERNATE-MOVE-OVER-EMPTY-SEQ
  fa () ALTERNATE-MOVE([]) = true

assert DISTRIBUTE-ALTERNATE-MOVE-OVER-SEQUENCE-CONCATENATE
  fa (S1: seq(integer), S2: seq(integer))
    ALTERNATE-MOVE(S1 ++ S2)
      = (ALTERNATE-MOVE(S1) and SHIFT-ALTERNATE-MOVE(S2, size(S1)))

assert
  DISTRIBUTE-ALTERNATE-MOVE-OVER-PREPEND-ELEMENT-TO-SEQUENCE
  fa (A, S: seq(integer))
    ALTERNATE-MOVE(prepend(S, A))
```

```
      = (ALTERNATE-MOVE([A]) and SHIFT-ALTERNATE-MOVE(S, 1))

assert
  DISTRIBUTE-ALTERNATE-MOVE-OVER-APPEND-ELEMENT-TO-SEQUENCE
  fa (A: integer, S: seq(integer))
    ALTERNATE-MOVE(append(S, A))
      = (ALTERNATE-MOVE(S) and SHIFT-ALTERNATE-MOVE([A], size(S)))

assert DISTRIBUTE-POSSIBLE-MOVE-OVER-SEQUENCE-CONCATENATE
  fa (S1: seq(tuple(integer, integer, integer)),
      S2: seq(tuple(integer, integer, integer)), K: any-type)
    POSSIBLE-MOVE(S1 ++ S2, K)
      = (POSSIBLE-MOVE(S1, K) and POSSIBLE-MOVE(S2, K))

assert DISTRIBUTE-POSSIBLE-MOVE-OVER-SINGLETON-SEQUENCE
  fa (M: any-type, L)
    POSSIBLE-MOVE([<M, M, L>], M)
      = (if M > 0 then true else false)

assert
  DISTRIBUTE-POSSIBLE-MOVE-OVER-PREPEND-ELEMENT-TO-SEQUENCE
  fa (A, S: seq(tuple(integer, integer, integer)), K: any-type)
    POSSIBLE-MOVE(prepend(S, A), K)
      = (POSSIBLE-MOVE([A], K) and POSSIBLE-MOVE(S, K))

assert
  DISTRIBUTE-POSSIBLE-MOVE-OVER-APPEND-ELEMENT-TO-SEQUENCE
  fa (A: tuple(integer, integer, integer),
      S: seq(tuple(integer, integer, integer)), K: any-type)
    POSSIBLE-MOVE(append(S, A), K)
      = (POSSIBLE-MOVE(S, K) and POSSIBLE-MOVE([A], K))

assert DISTRIBUTE-VISITED-STATES-OVER-EMPTY-SEQ
  fa (STATES: tuple(integer, integer, integer))
    VISITED-STATES([], STATES) = [STATES]

assert DISTRIBUTE-VISITED-STATES-OVER-SEQUENCE-CONCATENATE
  fa (S1: seq(integer), S2: seq(integer),
      STATES: tuple(integer, integer, integer))
    VISITED-STATES(S1 ++ S2, STATES)
      = VISITED-STATES(S1, STATES)
          ++ ALL-BUT-1ST-VISITED-STATES
               (S2, last(VISITED-STATES(S1, STATES)))

assert
```

```
  DISTRIBUTE-VISITED-STATES-OVER-PREPEND-ELEMENT-TO-SEQUENCE
  fa (A, S: seq(integer),
      STATES: tuple(integer, integer, integer))
    VISITED-STATES(prepend(S, A), STATES)
      = VISITED-STATES([A], STATES)
          ++ ALL-BUT-1ST-VISITED-STATES
               (S, last(VISITED-STATES([A], STATES)))

assert
  DISTRIBUTE-VISITED-STATES-OVER-APPEND-ELEMENT-TO-SEQUENCE
  fa (A: integer, S: seq(integer),
      STATES: tuple(integer, integer, integer))
    VISITED-STATES(append(S, A), STATES)
      = VISITED-STATES(S, STATES)
          ++ ALL-BUT-1ST-VISITED-STATES
               ([A], last(VISITED-STATES(S, STATES)))

assert DISTRIBUTE-NO-CANNIBALISM-OVER-SEQUENCE-CONCATENATE
  fa (S1: seq(tuple(integer, integer, integer)),
      S2: seq(tuple(integer, integer, integer)), K: integer)
    NO-CANNIBALISM(S1 ++ S2, K)
      = (NO-CANNIBALISM(S1, K) and NO-CANNIBALISM(S2, K))

assert DISTRIBUTE-NO-CANNIBALISM-OVER-SINGLETON-SEQUENCE
  fa (K: integer, L)
    NO-CANNIBALISM([<K, K, L>], K) = true

assert
  DISTRIBUTE-NO-CANNIBALISM-OVER-PREPEND-ELEMENT-TO-SEQUENCE
  fa (A, S: seq(tuple(integer, integer, integer)), K: integer)
    NO-CANNIBALISM(prepend(S, A), K)
      = (NO-CANNIBALISM([A], K) and NO-CANNIBALISM(S, K))

assert
  DISTRIBUTE-NO-CANNIBALISM-OVER-APPEND-ELEMENT-TO-SEQUENCE
  fa (A: tuple(integer, integer, integer),
      S: seq(tuple(integer, integer, integer)), K: integer)
    NO-CANNIBALISM(append(S, A), K)
      = (NO-CANNIBALISM([A], K) and NO-CANNIBALISM(S, K))

assert DISTRIBUTE-NO-MOVES-BACK-OVER-SINGLETON-SEQ
  fa (A) NO-MOVES-BACK([A]) = true

assert DISTRIBUTE-NO-MOVES-BACK-OVER-SEQUENCE-CONCATENATE
  fa (R: seq(tuple(integer, integer, integer)),
```

```
      S: seq(tuple(integer, integer, integer)))
    NO-MOVES-BACK(R ++ S)
      = (NO-MOVES-BACK(R) and NO-MOVES-BACK(S)
          and CROSS-NO-MOVES-BACK(R, S))

assert
  DISTRIBUTE-NO-MOVES-BACK-OVER-PREPEND-ELEMENT-TO-SEQUENCE
  fa (S: seq(tuple(integer, integer, integer)), A)
    NO-MOVES-BACK(prepend(S, A))
      = (NO-MOVES-BACK(S) and CROSS-NO-MOVES-BACK(S, [A]))

assert
  DISTRIBUTE-NO-MOVES-BACK-OVER-APPEND-ELEMENT-TO-SEQUENCE
  fa (S: seq(tuple(integer, integer, integer)),
      A: tuple(integer, integer, integer))
    NO-MOVES-BACK(append(S, A))
      = (NO-MOVES-BACK(S) and CROSS-NO-MOVES-BACK(S, [A]))

assert
  DISTRIBUTE-EVERYBODY-MADE-IT-OVER-SEQUENCE-CONCATENATE
  fa (S1: seq(tuple(integer, integer, integer)),
      S2: seq(tuple(integer, integer, integer)))
    EVERYBODY-MADE-IT(S1 ++ S2) = EVERYBODY-MADE-IT(S2)

assert
  DISTRIBUTE-EVERYBODY-MADE-IT-OVER-PREPEND-ELEMENT-TO-SEQUENCE
  fa (A, S: seq(tuple(integer, integer, integer)))
    EVERYBODY-MADE-IT(prepend(S, A)) = EVERYBODY-MADE-IT(S)

assert
  DISTRIBUTE-EVERYBODY-MADE-IT-OVER-APPEND-ELEMENT-TO-SEQUENCE
  fa (A, S)
    EVERYBODY-MADE-IT(append(S, A)) = EVERYBODY-MADE-IT([A])

%-----------------------------------------------------------------------
THEORY-RULES

"fa () ALTERNATE-MOVE([]) = true"
function
  MISSIONARIES-RULE-DISTRIBUTE-ALTERNATE-MOVE-OVER-EMPTY-SEQ-REWRITE
  () rb-compile-simplification-equality
       DISTRIBUTE-ALTERNATE-MOVE-OVER-EMPTY-SEQ

"fa (S1, S2)
  ALTERNATE-MOVE(S1 ++ S2)
```

```
    = (ALTERNATE-MOVE(S1) and SHIFT-ALTERNATE-MOVE(S2, size(S1)))"
function
  MISSIONARIES-RULE-DISTRIBUTE-ALTERNATE-MOVE-OVER-SEQUENCE-CONCATENATE-REWRITE
  () rb-compile-simplification-equality
       DISTRIBUTE-ALTERNATE-MOVE-OVER-SEQUENCE-CONCATENATE

"fa (A, S)
  ALTERNATE-MOVE(prepend(S, A))
    = (ALTERNATE-MOVE([A]) and SHIFT-ALTERNATE-MOVE(S, 1))"
function
  MISSIONARIES-RULE-DISTRIBUTE-ALTERNATE-MOVE-OVER-PREPEND-ELEMENT-TO-SEQUENCE-REWR
  () rb-compile-simplification-equality
       DISTRIBUTE-ALTERNATE-MOVE-OVER-PREPEND-ELEMENT-TO-SEQUENCE

"fa (A, S)
  ALTERNATE-MOVE(append(S, A))
    = (ALTERNATE-MOVE(S) and SHIFT-ALTERNATE-MOVE([A], size(S)))"
function
  MISSIONARIES-RULE-DISTRIBUTE-ALTERNATE-MOVE-OVER-APPEND-ELEMENT-TO-SEQUENCE-REWRI
  () rb-compile-simplification-equality
       DISTRIBUTE-ALTERNATE-MOVE-OVER-APPEND-ELEMENT-TO-SEQUENCE

"fa (S1, S2, K)
  POSSIBLE-MOVE(S1 ++ S2, K)
    = (POSSIBLE-MOVE(S1, K) and POSSIBLE-MOVE(S2, K))"
function
  MISSIONARIES-RULE-DISTRIBUTE-POSSIBLE-MOVE-OVER-SEQUENCE-CONCATENATE-REWRITE
  () rb-compile-simplification-equality
       DISTRIBUTE-POSSIBLE-MOVE-OVER-SEQUENCE-CONCATENATE

"fa (A, S, K)
  POSSIBLE-MOVE(prepend(S, A), K)
    = (POSSIBLE-MOVE([A], K) and POSSIBLE-MOVE(S, K))"
function
  MISSIONARIES-RULE-DISTRIBUTE-POSSIBLE-MOVE-OVER-PREPEND-ELEMENT-TO-SEQUENCE-REWRI
  () rb-compile-simplification-equality
       DISTRIBUTE-POSSIBLE-MOVE-OVER-PREPEND-ELEMENT-TO-SEQUENCE

"fa (M, L)
  POSSIBLE-MOVE([<M, M, L>], M)
    = (if M > 0 then true else false)"
function
  MISSIONARIES-RULE-DISTRIBUTE-POSSIBLE-MOVE-OVER-SINGLETON-SEQUENCE-REWRITE
  () rb-compile-simplification-equality
       DISTRIBUTE-POSSIBLE-MOVE-OVER-SINGLETON-SEQUENCE
```

```
"fa (A, S, K)
  POSSIBLE-MOVE(append(S, A), K)
    = (POSSIBLE-MOVE(S, K) and POSSIBLE-MOVE([A], K))"
function
  MISSIONARIES-RULE-DISTRIBUTE-POSSIBLE-MOVE-OVER-APPEND-ELEMENT-TO-SEQUENCE-REWRIT
  () rb-compile-simplification-equality
       DISTRIBUTE-POSSIBLE-MOVE-OVER-APPEND-ELEMENT-TO-SEQUENCE

"fa (STATES) VISITED-STATES([], STATES) = [STATES]"
function
  MISSIONARIES-RULE-DISTRIBUTE-VISITED-STATES-OVER-EMPTY-SEQ
  () rb-compile-simplification-equality
       DISTRIBUTE-VISITED-STATES-OVER-EMPTY-SEQ

"fa (S1, S2, STATES)
  VISITED-STATES(S1 ++ S2, STATES)
    = VISITED-STATES(S1, STATES)
        ++ VISITED-STATES(S2, last(VISITED-STATES(S1, STATES)))"
function
  MISSIONARIES-RULE-DISTRIBUTE-VISITED-STATES-OVER-SEQUENCE-CONCATENATE-REWRITE
  () rb-compile-simplification-equality
       DISTRIBUTE-VISITED-STATES-OVER-SEQUENCE-CONCATENATE

"fa (A, S, STATES)
  VISITED-STATES(prepend(S, A), STATES)
    = VISITED-STATES([A], STATES)
        ++ VISITED-STATES(S, last(VISITED-STATES([A], STATES)))"
function
  MISSIONARIES-RULE-DISTRIBUTE-VISITED-STATES-OVER-PREPEND-ELEMENT-TO-SEQUENCE-REWR
  () rb-compile-simplification-equality
       DISTRIBUTE-VISITED-STATES-OVER-PREPEND-ELEMENT-TO-SEQUENCE

"fa (A, S, STATES)
  VISITED-STATES(append(S, A), STATES)
    = VISITED-STATES(S, STATES)
        ++ VISITED-STATES([A], last(VISITED-STATES(S, STATES)))"
function
  MISSIONARIES-RULE-DISTRIBUTE-VISITED-STATES-OVER-APPEND-ELEMENT-TO-SEQUENCE-REWRI
  () rb-compile-simplification-equality
       DISTRIBUTE-VISITED-STATES-OVER-APPEND-ELEMENT-TO-SEQUENCE

"fa (S1, S2, K)
  NO-CANNIBALISM(S1 ++ S2, K)
    = (NO-CANNIBALISM(S1, K) and NO-CANNIBALISM(S2, K))"
```

```
function
  MISSIONARIES-RULE-DISTRIBUTE-NO-CANNIBALISM-OVER-SEQUENCE-CONCATENATE-REWRITE
  () rb-compile-simplification-equality
       DISTRIBUTE-NO-CANNIBALISM-OVER-SEQUENCE-CONCATENATE

"fa (A, S, K)
  NO-CANNIBALISM(prepend(S, A), K)
    = (NO-CANNIBALISM([A], K) and NO-CANNIBALISM(S, K))"
function
  MISSIONARIES-RULE-DISTRIBUTE-NO-CANNIBALISM-OVER-PREPEND-ELEMENT-TO-SEQUENCE-REWR
  () rb-compile-simplification-equality
       DISTRIBUTE-NO-CANNIBALISM-OVER-PREPEND-ELEMENT-TO-SEQUENCE

"fa (A, S, K)
  NO-CANNIBALISM(append(S, A), K)
    = (NO-CANNIBALISM([A], K) and NO-CANNIBALISM(S, K))"
function
  MISSIONARIES-RULE-DISTRIBUTE-NO-CANNIBALISM-OVER-APPEND-ELEMENT-TO-SEQUENCE-REWRI
  () rb-compile-simplification-equality
       DISTRIBUTE-NO-CANNIBALISM-OVER-APPEND-ELEMENT-TO-SEQUENCE

"fa (K, L) NO-CANNIBALISM([<K, K, L>], K) = true"
function
  MISSIONARIES-RULE-DISTRIBUTE-NO-CANNIBALISM-OVER-SINGLETON-SEQUENCE-REWRITE
  () rb-compile-simplification-equality
       DISTRIBUTE-NO-CANNIBALISM-OVER-SINGLETON-SEQUENCE

"fa (A) NO-MOVES-BACK([A]) = true"
function
  MISSIONARIES-RULE-DISTRIBUTE-NO-MOVES-BACK-OVER-SINGLETON-SEQ-REWRITE
  () rb-compile-simplification-equality
       DISTRIBUTE-NO-MOVES-BACK-OVER-SINGLETON-SEQ

"fa (R, S)
  NO-MOVES-BACK(R ++ S)
    = (NO-MOVES-BACK(R) and NO-MOVES-BACK(S)
         and CROSS-NO-MOVES-BACK(R, S))"
function
  MISSIONARIES-RULE-DISTRIBUTE-NO-MOVES-BACK-OVER-SEQUENCE-CONCATENATE-REWRITE
  () rb-compile-simplification-equality
       DISTRIBUTE-NO-MOVES-BACK-OVER-SEQUENCE-CONCATENATE

"fa (S, A)
  NO-MOVES-BACK(prepend(S, A))
    = (NO-MOVES-BACK(S) and CROSS-NO-MOVES-BACK(S, [A]))"
```

```
function
  MISSIONARIES-RULE-DISTRIBUTE-NO-MOVES-BACK-OVER-PREPEND-ELEMENT-TO-SEQUENCE-REWRI
  () rb-compile-simplification-equality
       DISTRIBUTE-NO-MOVES-BACK-OVER-PREPEND-ELEMENT-TO-SEQUENCE

"fa (S, A)
  NO-MOVES-BACK(append(S, A))
    = (NO-MOVES-BACK(S) and CROSS-NO-MOVES-BACK(S, [A]))"
function
  MISSIONARIES-RULE-DISTRIBUTE-NO-MOVES-BACK-OVER-APPEND-ELEMENT-TO-SEQUENCE-REWRIT
  () rb-compile-simplification-equality
       DISTRIBUTE-NO-MOVES-BACK-OVER-APPEND-ELEMENT-TO-SEQUENCE

"fa (S1, S2)
  EVERYBODY-MADE-IT(S1 ++ S2) = EVERYBODY-MADE-IT(S2)"
function
  MISSIONARIES-RULE-DISTRIBUTE-EVERYBODY-MADE-IT-OVER-SEQUENCE-CONCATENATE-REWRITE
  () rb-compile-simplification-equality
       DISTRIBUTE-EVERYBODY-MADE-IT-OVER-SEQUENCE-CONCATENATE

"fa (A, S)
  EVERYBODY-MADE-IT(prepend(S, A)) = EVERYBODY-MADE-IT(S)"
function
  MISSIONARIES-RULE-DISTRIBUTE-EVERYBODY-MADE-IT-OVER-PREPEND-ELEMENT-TO-SEQUENCE-R
  () rb-compile-simplification-equality
       DISTRIBUTE-EVERYBODY-MADE-IT-OVER-PREPEND-ELEMENT-TO-SEQUENCE

"fa (A, S)
  EVERYBODY-MADE-IT(append(S, A)) = EVERYBODY-MADE-IT([A])"
function
  MISSIONARIES-RULE-DISTRIBUTE-EVERYBODY-MADE-IT-OVER-APPEND-ELEMENT-TO-SEQUENCE-RE
  () rb-compile-simplification-equality
       DISTRIBUTE-EVERYBODY-MADE-IT-OVER-APPEND-ELEMENT-TO-SEQUENCE

%----------------------------------------------------------------------
THEORY-MISC-LAWS

%----------------------------------------------------------------------
THEORY-MISC-DEFS

%----------------------------------------------------------------------
THEORY-MISC-RULES

%----------------------------------------------------------------------
THEORY-MISC-FORMS
```

```
%-----------------------------------------------------------------------
end-theory
```

6.2 Excerpt of Rainbow Inference to Reduce the MC PROBLEM to to the gs-theory *gs-sequences-over-finite-set*

Rainbow is the theorem prover in KIDS.

```
Selecting a standard global search theory:
GS-SEQUENCES-OVER-FINITE-DOMAIN

Creating a specialized global search theory for the problem
by deriving a complete-reduction substitution.
Performing forward inference to acquire new assumptions
Depth bound:

Entering Rainbow to derive a CONSEQUENT
Target variables: var MC: seq(integer) constant K: integer
Assumptions: none
Goal: EVERYBODY-MADE-IT(VISITED-STATES(MC, <K, K, 1>))
  & NO-MOVES-BACK(VISITED-STATES(MC, <K, K, 1>))
  & NO-CANNIBALISM(VISITED-STATES(MC, <K, K, 1>), K)
  & POSSIBLE-MOVE(VISITED-STATES(MC, <K, K, 1>), K)
  & ALTERNATE-MOVE(MC) & range(MC) subset {1 .. 10} & 1 <= K
Initial terms at depth 0
1. true
2. 1 <= K
3. ALTERNATE-MOVE(MC)
4. range(MC) subset {1 .. 10}
5. NO-MOVES-BACK(VISITED-STATES(MC, <K, K, 1>))
6. EVERYBODY-MADE-IT(VISITED-STATES(MC, <K, K, 1>))
7. POSSIBLE-MOVE(VISITED-STATES(MC, <K, K, 1>), K)
8. NO-CANNIBALISM(VISITED-STATES(MC, <K, K, 1>), K)

Derived terms at depth 1
1. size(range(MC)) <= size({1 .. 10})
2. setdiff(range(MC), {1 .. 10}) = {}


Derived terms at depth 2
```

```
1. size(range(MC)) <= 10

All derived terms:
1. true
2. 1 <= K
3. ALTERNATE-MOVE(MC)
4. range(MC) subset {1 .. 10}
5. size(range(MC)) <= size({1 .. 10})
6. size(range(MC)) <= 10
7. setdiff(range(MC), {1 .. 10}) = {}
8. NO-MOVES-BACK(VISITED-STATES(MC, <K, K, 1>))
9. EVERYBODY-MADE-IT(VISITED-STATES(MC, <K, K, 1>))
10. POSSIBLE-MOVE(VISITED-STATES(MC, <K, K, 1>), K)
11. NO-CANNIBALISM(VISITED-STATES(MC, <K, K, 1>), K)

Entering Rainbow to derive an ANTECEDENT
Target variables:
Assumptions:
1. setdiff(range(MC), {1 .. 10}) = {}
2. NO-CANNIBALISM(VISITED-STATES(MC, <K, K, 1>), K)
3. POSSIBLE-MOVE(VISITED-STATES(MC, <K, K, 1>), K)
4. EVERYBODY-MADE-IT(VISITED-STATES(MC, <K, K, 1>))
5. NO-MOVES-BACK(VISITED-STATES(MC, <K, K, 1>))
6. size(range(MC)) <= 10
7. size(range(MC)) <= size({1 .. 10})
8. range(MC) subset {1 .. 10}
9. ALTERNATE-MOVE(MC)
10. 1 <= K
Goal: range(MC) subset S
Subgoal: range(MC) subset S
Subgoal: true
RAINBOW result: true

Substitution: {S->{1 .. 10}}


Inferred i/o-relation: true
Complete-reduction substitution: {S->{1 .. 10}}
```

6.3 Excerpt of Rainbow Inference to Derive Filters

New global search theory incorporating the substitution:

```
(operator-specification MC-TOP number-of-solutions ALL
  input-types integer output-types seq(integer) input-vars K
  output-vars MC input-condition K >= 1
  output-condition
    range(MC) subset {1 .. 10} & ALTERNATE-MOVE(MC)
      & POSSIBLE-MOVE(VISITED-STATES(MC, <K, K, 1>), K)
      & NO-CANNIBALISM(VISITED-STATES(MC, <K, K, 1>), K)
      & NO-MOVES-BACK(VISITED-STATES(MC, <K, K, 1>))
      & EVERYBODY-MADE-IT(VISITED-STATES(MC, <K, K, 1>)))
```

Deriving a filter on subspaces by finding a necessary condition on the existence of solutions in a subspace:

```
Entering Rainbow to derive a CONSEQUENT
Target variables: constant K: integer var V
Assumptions:
1. 1 <= K
Goal: range(V) subset {1 .. 10} & MC = concat(V, Y-197)
  & EVERYBODY-MADE-IT(VISITED-STATES(MC, <K, K, 1>))
  & NO-MOVES-BACK(VISITED-STATES(MC, <K, K, 1>))
  & NO-CANNIBALISM(VISITED-STATES(MC, <K, K, 1>), K)
  & POSSIBLE-MOVE(VISITED-STATES(MC, <K, K, 1>), K)
  & ALTERNATE-MOVE(MC) & range(MC) subset {1 .. 10}
Initial terms at depth 0
1. true
2. range(V) subset {1 .. 10}
3. ALTERNATE-MOVE(MC)
4. range(MC) subset {1 .. 10}
5. MC = concat(V, Y-197)
6. NO-MOVES-BACK(VISITED-STATES(MC, <K, K, 1>))
7. EVERYBODY-MADE-IT(VISITED-STATES(MC, <K, K, 1>))
8. POSSIBLE-MOVE(VISITED-STATES(MC, <K, K, 1>), K)
9. NO-CANNIBALISM(VISITED-STATES(MC, <K, K, 1>), K)

Derived terms at depth 1
1. ALTERNATE-MOVE(V)
2. range(Y-197) subset {1 .. 10}
3. size(range(V)) <= size({1 .. 10})
4. setdiff(range(V), {1 .. 10}) = {}
5. SHIFT-ALTERNATE-MOVE(Y-197, size(V))
6. NO-MOVES-BACK(VISITED-STATES(V, <K, K, 1>))
7. POSSIBLE-MOVE(VISITED-STATES(V, <K, K, 1>), K)
8. NO-CANNIBALISM(VISITED-STATES(V, <K, K, 1>), K)
9. NO-MOVES-BACK
  (ALL-BUT-1ST-VISITED-STATES
```

```
     (Y-197, last(VISITED-STATES(V, <K, K, 1>))))
10. EVERYBODY-MADE-IT
  (ALL-BUT-1ST-VISITED-STATES
     (Y-197, last(VISITED-STATES(V, <K, K, 1>))))
11. POSSIBLE-MOVE
  (ALL-BUT-1ST-VISITED-STATES
     (Y-197, last(VISITED-STATES(V, <K, K, 1>))),
   K)
12. NO-CANNIBALISM
  (ALL-BUT-1ST-VISITED-STATES
     (Y-197, last(VISITED-STATES(V, <K, K, 1>))),
   K)
13. size(range(MC)) <= size({1 .. 10})
14. setdiff(range(MC), {1 .. 10}) = {}
15. CROSS-NO-MOVES-BACK
  (VISITED-STATES(V, <K, K, 1>),
   ALL-BUT-1ST-VISITED-STATES
     (Y-197, last(VISITED-STATES(V, <K, K, 1>))))

Derived terms at depth 2
1. size(range(V)) <= 10
2. size(range(Y-197)) <= size({1 .. 10})
3. setdiff(range(Y-197), {1 .. 10}) = {}
4. size(range(Y-197) union range(V)) <= size({1 .. 10})
5. size(range(MC)) <= 10
6. setdiff(range(V), {1 .. 10})
  = setdiff(range(MC), {1 .. 10})
7. setdiff(range(MC), {1 .. 10})
  = setdiff(range(V), {1 .. 10})

All derived terms:
1. true
2. ALTERNATE-MOVE(V)
3. range(V) subset {1 .. 10}
4. range(Y-197) subset {1 .. 10}
5. size(range(V)) <= size({1 .. 10})
6. size(range(V)) <= 10
7. setdiff(range(V), {1 .. 10}) = {}
8. SHIFT-ALTERNATE-MOVE(Y-197, size(V))
9. size(range(Y-197)) <= size({1 .. 10})
10. setdiff(range(Y-197), {1 .. 10}) = {}
11. NO-MOVES-BACK(VISITED-STATES(V, <K, K, 1>))
12. size(range(Y-197) union range(V)) <= size({1 .. 10})
13. POSSIBLE-MOVE(VISITED-STATES(V, <K, K, 1>), K)
14. NO-CANNIBALISM(VISITED-STATES(V, <K, K, 1>), K)
```

```
15. ALTERNATE-MOVE(MC)
16. NO-MOVES-BACK
  (ALL-BUT-1ST-VISITED-STATES
     (Y-197, last(VISITED-STATES(V, <K, K, 1>))))
17. EVERYBODY-MADE-IT
  (ALL-BUT-1ST-VISITED-STATES
     (Y-197, last(VISITED-STATES(V, <K, K, 1>))))
18. range(MC) subset {1 .. 10}
19. POSSIBLE-MOVE
  (ALL-BUT-1ST-VISITED-STATES
     (Y-197, last(VISITED-STATES(V, <K, K, 1>))),
   K)
20. NO-CANNIBALISM
  (ALL-BUT-1ST-VISITED-STATES
     (Y-197, last(VISITED-STATES(V, <K, K, 1>))),
   K)
21. size(range(MC)) <= size({1 .. 10})
22. size(range(MC)) <= 10
23. setdiff(range(MC), {1 .. 10}) = {}
24. MC = concat(V, Y-197)
25. NO-MOVES-BACK(VISITED-STATES(MC, <K, K, 1>))
26. EVERYBODY-MADE-IT(VISITED-STATES(MC, <K, K, 1>))
27. setdiff(range(V), {1 .. 10})
  = setdiff(range(MC), {1 .. 10})
28. setdiff(range(MC), {1 .. 10})
  = setdiff(range(V), {1 .. 10})
29. POSSIBLE-MOVE(VISITED-STATES(MC, <K, K, 1>), K)
30. NO-CANNIBALISM(VISITED-STATES(MC, <K, K, 1>), K)
31. CROSS-NO-MOVES-BACK
  (VISITED-STATES(V, <K, K, 1>),
   ALL-BUT-1ST-VISITED-STATES
     (Y-197, last(VISITED-STATES(V, <K, K, 1>))))
```

From all the derived terms, KIDS proposes to the user as filters the ones that are a function of the input variable k and the space descriptor V. Several of the filters proposed by KIDS are redundant or don't have any pruning impact - a good example of that is the filter "true". From the filters proposed by KIDS, the significant ones were selected:

```
NO-CANNIBALISM(VISITED-STATES(V, <K, K, 1>), K)
  & POSSIBLE-MOVE(VISITED-STATES(V, <K, K, 1>), K)
  & NO-MOVES-BACK(VISITED-STATES(V, <K, K, 1>))
  & ALTERNATE-MOVE(V)
```

```
New global search theory incorporating the necessary
filter:(operator-specification MC-TOP number-of-solutions ALL
  input-types integer output-types seq(integer) input-vars K
  output-vars MC input-condition K >= 1
  output-condition
    range(MC) subset {1 .. 10} & ALTERNATE-MOVE(MC)
      & POSSIBLE-MOVE(VISITED-STATES(MC, <K, K, 1>), K)
      & NO-CANNIBALISM(VISITED-STATES(MC, <K, K, 1>), K)
      & NO-MOVES-BACK(VISITED-STATES(MC, <K, K, 1>))
      & EVERYBODY-MADE-IT(VISITED-STATES(MC, <K, K, 1>)))

Assembling a global-search algorithm
Choosing MULTILINEAR-RECURSIVE-WITH-DISJOINT-UNION as the assembling
 scheme done.
```

6.3.1 Context Independent Simplification

The following is an example of a context-independent simplification performed by KIDS. It mainly uses the distributive laws regarding the different predicates that define the output condition for the base case [ ], where they simplify to *true*.

The boldface code in the function MC-TOP indicates the block of code that was selected to be simplified using the context independent simplification:

```
Stpe 3:Context-Independent-Fast
  expression-to-be-simplified: if ## & ## & ## & ## then MC-TOP-AUX(##,...
```

function MC-TOP
(var ##p attribute false VARIABLE? K: integer | K ≥1)
returns
(MC-TOP-SET: set(seq(integer)) |
MC-TOP-SET
= {MC | (MC: seq(integer))
range(MC) ⊆{1 ..10}
∧ALTERNATE-MOVE(MC)
∧POSSIBLE-MOVE
(VISITED-STATES(MC, ⟨K, K, 1⟩), K)
∧NO-CANNIBALISM
(VISITED-STATES(MC, ⟨K, K, 1⟩), K)

```
                ∧NO-MOVES-BACK(VISITED-STATES(MC, ⟨K, K, 1⟩))
                ∧EVERYBODY-MADE-IT
                      (VISITED-STATES(MC, ⟨K, K, 1⟩))})
    = if NO-CANNIBALISM
           (VISITED-STATES([], ⟨K, K, 1⟩), K)
           ∧POSSIBLE-MOVE
                 (VISITED-STATES([], ⟨K, K, 1⟩), K)
           ∧NO-MOVES-BACK(VISITED-STATES([], ⟨K, K, 1⟩))
           ∧ALTERNATE-MOVE([])
         then MC-TOP-AUX(K, [])
        else {}
```

After applying the context independent simplification to the boldface code in the function MC-TOP, the resulting code looks like:

```
function MC-TOP
    (var ##p attribute false VARIABLE? K: integer | K ≥1)
    returns
      (MC-TOP-SET: set(seq(integer)) |
         MC-TOP-SET
           = {MC | (MC: seq(integer))
              range(MC) ⊆{1 ..10}
                ∧ALTERNATE-MOVE(MC)
                ∧POSSIBLE-MOVE
                      (VISITED-STATES(MC, ⟨K, K, 1⟩), K)
                ∧NO-CANNIBALISM
                      (VISITED-STATES(MC, ⟨K, K, 1⟩), K)
                ∧NO-MOVES-BACK(VISITED-STATES(MC, ⟨K, K, 1⟩))
                ∧EVERYBODY-MADE-IT
                      (VISITED-STATES(MC, ⟨K, K, 1⟩))})
    = if 0 <Kthen MC-TOP-AUX(K, []) else {}
```

6.3.2 Context Independent Simplification

The following is an example of a context-independent simplification performed by KIDS. It mainly uses the distributive laws regarding the different predicates that define the output condition for the base case [ ], where they simplify to *true*.

The boldface code in the function MC-TOP indicates the block of code that was selected to be simplified using the context independent simplification:

```
Stpe 3:Context-Independent-Fast
  expression-to-be-simplified: if ## & ## & ## & ## then MC-TOP-AUX(##,...
```

```
function MC-TOP
    (var ##p attribute false VARIABLE? K: integer | K ≥1)
    returns
      (MC-TOP-SET: set(seq(integer)) |
         MC-TOP-SET
           = {MC | (MC: seq(integer))
                range(MC) ⊆{1 ..10}
                  ∧ALTERNATE-MOVE(MC)
                  ∧POSSIBLE-MOVE
                        (VISITED-STATES(MC, ⟨K, K, 1⟩), K)
                  ∧NO-CANNIBALISM
                        (VISITED-STATES(MC, ⟨K, K, 1⟩), K)
                  ∧NO-MOVES-BACK(VISITED-STATES(MC, ⟨K, K, 1⟩))
                  ∧EVERYBODY-MADE-IT
                        (VISITED-STATES(MC, ⟨K, K, 1⟩))})
    = if NO-CANNIBALISM
            (VISITED-STATES([], ⟨K, K, 1⟩), K)
            ∧POSSIBLE-MOVE
                  (VISITED-STATES([], ⟨K, K, 1⟩), K)
            ∧NO-MOVES-BACK(VISITED-STATES([], ⟨K, K, 1⟩))
            ∧ALTERNATE-MOVE([])
        then MC-TOP-AUX(K, [])
       else {}
```

After applying the context independent simplification to the boldface code in the function MC-TOP, the resulting code looks like:

```
function MC-TOP
    (var ##p attribute false VARIABLE? K: integer | K ≥1)
    returns
      (MC-TOP-SET: set(seq(integer)) |
         MC-TOP-SET
           = {MC | (MC: seq(integer))
                range(MC) ⊆{1 ..10}
                  ∧ALTERNATE-MOVE(MC)
                  ∧POSSIBLE-MOVE
                        (VISITED-STATES(MC, ⟨K, K, 1⟩), K)
                  ∧NO-CANNIBALISM
                        (VISITED-STATES(MC, ⟨K, K, 1⟩), K)
                  ∧NO-MOVES-BACK(VISITED-STATES(MC, ⟨K, K, 1⟩))
                  ∧EVERYBODY-MADE-IT
                        (VISITED-STATES(MC, ⟨K, K, 1⟩))})
    = if 0 <K then MC-TOP-AUX(K, []) else {}
```

6.4 Context Dependent Simplification

In the following example, the boldface code in the function MC-TOP-AUX is simplified to just a single predicate EVERYBODY-MADE-IT, since the other conjuncts are *true* in the context - they correspond to the filter.

The boldface code in the function MC-TOP-AUX indicates the block of code that was selected to be simplified using the context independent simplification:

```
Context-Dependent-Forward-0-Backward-4
  expression-to-be-simplified: range(##) subset {## .. ##} & ALTERNATE-MOV...
```

function MC-TOP-AUX
(var ##p attribute false VARIABLE? K: integer,
var ##p attribute false VARIABLE? V: seq(integer)
|
K ≥1 ∧range(V) ⊆{1 ..10}
∧ALTERNATE-MOVE(V)
∧NO-MOVES-BACK(VISITED-STATES(V, ⟨K, K, 1⟩))
∧POSSIBLE-MOVE(VISITED-STATES(V, ⟨K, K, 1⟩), K)
∧NO-CANNIBALISM
(VISITED-STATES(V, ⟨K, K, 1⟩), K))
returns
(MC-TOP-SET: set(seq(integer)) |
MC-TOP-SET
= {MC | (MC: seq(integer))
EXTENDS(MC, V)
∧EVERYBODY-MADE-IT
(VISITED-STATES(MC, ⟨K, K, 1⟩))
∧NO-MOVES-BACK(VISITED-STATES(MC, ⟨K, K, 1⟩))
∧NO-CANNIBALISM
(VISITED-STATES(MC, ⟨K, K, 1⟩), K)
∧POSSIBLE-MOVE
(VISITED-STATES(MC, ⟨K, K, 1⟩), K)
∧ALTERNATE-MOVE(MC)
∧range(MC) ⊆{1 ..10}})
= {V | ()
range(V) ⊆{1 ..10}
∧ALTERNATE-MOVE(V)
∧POSSIBLE-MOVE(VISITED-STATES(V, ⟨K, K, 1⟩), K)
∧NO-CANNIBALISM
(VISITED-STATES(V, ⟨K, K, 1⟩), K)

```
        ∧NO-MOVES-BACK(VISITED-STATES(V, ⟨K, K, 1⟩))
        ∧EVERYBODY-MADE-IT
              (VISITED-STATES(V, ⟨K, K, 1⟩))}
    ∪!reduce (...)
```

After applying the context independent simplification to the boldface code in function MC-TOP-AUX, the resulting code looks like:

```
function MC-TOPAUX
    (var ##p attribute false VARIABLE? K: integer,
     var ##p attribute false VARIABLE? V: seq(integer)
      |
        K ≥1 ∧range(V) ⊆{1 ..10}
          ∧ALTERNATE-MOVE(V)
          ∧NO-MOVES-BACK(VISITED-STATES(V, ⟨K, K, 1⟩))
          ∧POSSIBLE-MOVE(VISITED-STATES(V, ⟨K, K, 1⟩), K)
          ∧NO-CANNIBALISM
                (VISITED-STATES(V, ⟨K, K, 1⟩), K))
    returns
      (MC-TOP-SET: set(seq(integer)) |
        MC-TOP-SET
          = {MC | (MC: seq(integer))
              EXTENDS(MC, V)
                ∧EVERYBODY-MADE-IT
                      (VISITED-STATES(MC, ⟨K, K, 1⟩))
                ∧NO-MOVES-BACK(VISITED-STATES(MC, ⟨K, K, 1⟩))
                ∧NO-CANNIBALISM
                      (VISITED-STATES(MC, ⟨K, K, 1⟩), K)
                ∧POSSIBLE-MOVE
                      (VISITED-STATES(MC, ⟨K, K, 1⟩), K)
                ∧ALTERNATE-MOVE(MC)
                ∧range(MC) ⊆{1 ..10}})
    = {V | ()
        EVERYBODY-MADE-IT(VISITED-STATES(V, ⟨K, K, 1⟩))}
        ∪!reduce (...)
```

6.5 Finite Difference of VISITED-STATES(V, <K,K,1>)

```
Step 10: Finite Difference
  expr-to-maintain: VISITED-STATES(V, <K, K, 1>)
  maintained-variable-name: 'VS
  special-purpose?: false
```

```
mapping-construct-preference: 'MAPFORMER
```

```
function MC-TOP-AUX
    (var ##p attribute false VARIABLE? K: integer,
     var ##p attribute false VARIABLE? V: seq(integer)
       |
         K ≥1 ∧range(V) ⊆{1 ..10}
           ∧ALTERNATE-MOVE(V)
           ∧NO-MOVES-BACK(VISITED-STATES(V, ⟨K, K, 1⟩))
           ∧POSSIBLE-MOVE(VISITED-STATES(V, ⟨K, K, 1⟩), K)
           ∧NO-CANNIBALISM
                 (VISITED-STATES(V, ⟨K, K, 1⟩), K))
    returns
      (MC-TOP-SET: set(seq(integer)) |
         MC-TOP-SET
           = {MC | (MC: seq(integer))
              EXTENDS(MC, V)
                ∧EVERYBODY-MADE-IT
                      (VISITED-STATES(MC, ⟨K, K, 1⟩))
                ∧NO-MOVES-BACK(VISITED-STATES(MC, ⟨K, K, 1⟩))
                ∧NO-CANNIBALISM
                      (VISITED-STATES(MC, ⟨K, K, 1⟩), K)
                ∧POSSIBLE-MOVE
                      (VISITED-STATES(MC, ⟨K, K, 1⟩), K)
                ∧ALTERNATE-MOVE(MC)
                ∧range(MC) ⊆{1 ..10}})
    = {V | ()
        EVERYBODY-MADE-IT(VISITED-STATES(V, ⟨K, K, 1⟩))}
        ∪!reduce
                (UNION!,
                 {MC-TOP-AUX(K, append(V, I)) | (I: integer)
                   NO-CANNIBALISM
                     (ALL-BUT-1ST-VISITED-STATES
                        ([I], last(VISITED-STATES(V, ⟨K, K, 1⟩))),
                      K)
                     ∧POSSIBLE-MOVE
                           (ALL-BUT-1ST-VISITED-STATES
                              ([I],
                               last(VISITED-STATES(V, ⟨K, K, 1⟩))),
                            K)
                     ∧CROSS-NO-MOVES-BACK
                           (VISITED-STATES(V, ⟨K, K, 1⟩),
```

```
                    ALL-BUT-1ST-VISITED-STATES
                      ([I],
                       last(VISITED-STATES(V, ⟨K, K, 1⟩)))))
            ∧NO-MOVES-BACK
                  (ALL-BUT-1ST-VISITED-STATES
                      ([I],
                       last(VISITED-STATES(V, ⟨K, K, 1⟩)))))
            ∧SHIFT-ALTERNATE-MOVE([I], size(V))
            ∧I ∈{1 ..10}})
```

After the finite difference operation of VISITED-STATES(V, <K,K,1>) into the variable VS, the new code produced by KIDS:

```
function MC-TOP-AUX
    (var ##p attribute false VARIABLE? K: integer,
     var ##p attribute false VARIABLE? V: seq(integer),
     var ##p attribute false VARIABLE? VS
       : seq(tuple(integer, integer, integer))
       |
       SEQEQUAL(VS, VISITED-STATES(V, ⟨K, K, 1⟩))
         ∧NO-CANNIBALISM
               (VISITED-STATES(V, ⟨K, K, 1⟩), K)
         ∧POSSIBLE-MOVE(VISITED-STATES(V, ⟨K, K, 1⟩), K)
         ∧NO-MOVES-BACK(VISITED-STATES(V, ⟨K, K, 1⟩))
         ∧ALTERNATE-MOVE(V)
         ∧range(V) ⊆{1 ..10}∧K ≥1)
  returns
    (MC-TOP-SET: set(seq(integer)) |
       MC-TOP-SET
         = {MC | (MC: seq(integer))
             EXTENDS(MC, V)
               ∧EVERYBODY-MADE-IT
                     (VISITED-STATES(MC, ⟨K, K, 1⟩))
               ∧NO-MOVES-BACK(VISITED-STATES(MC, ⟨K, K, 1⟩))
               ∧NO-CANNIBALISM
                     (VISITED-STATES(MC, ⟨K, K, 1⟩), K)
               ∧POSSIBLE-MOVE
                     (VISITED-STATES(MC, ⟨K, K, 1⟩), K)
               ∧ALTERNATE-MOVE(MC)
               ∧range(MC) ⊆{1 ..10}})
  = {V | () EVERYBODY-MADE-IT(VS)}
      U!reduce
              (UNION!,
               {MC-TOP-AUX
                     (K, append(V, I),
```

```
                    VS ++ALL-BUT-1ST-VISITED-STATES
                                    ([I], last(VS)))
                | (I: integer)
                NO-CANNIBALISM
                  (ALL-BUT-1ST-VISITED-STATES([I], last(VS)), K)
                  ∧POSSIBLE-MOVE
                        (ALL-BUT-1ST-VISITED-STATES([I], last(VS)), K)
                  ∧CROSS-NO-MOVES-BACK
                        (VS, ALL-BUT-1ST-VISITED-STATES([I], last(VS)))
                  ∧NO-MOVES-BACK
                        (ALL-BUT-1ST-VISITED-STATES([I], last(VS)))
                  ∧SHIFT-ALTERNATE-MOVE([I], size(V))
                  ∧I ∈{1 ..10}})

function MC-TOP
    (var ##p attribute false VARIABLE? K: integer | 1 ≤K)
    returns
      (MC-TOP-SET: set(seq(integer)) |
         MC-TOP-SET
           = {MC | (MC: seq(integer))
               range(MC) ⊆{1 ..10}
                 ∧ALTERNATE-MOVE(MC)
                 ∧POSSIBLE-MOVE
                       (VISITED-STATES(MC, ⟨K, K, 1⟩), K)
                 ∧NO-CANNIBALISM
                       (VISITED-STATES(MC, ⟨K, K, 1⟩), K)
                 ∧NO-MOVES-BACK(VISITED-STATES(MC, ⟨K, K, 1⟩))
                 ∧EVERYBODY-MADE-IT
                       (VISITED-STATES(MC, ⟨K, K, 1⟩))})
    = MC-TOP-AUX(K, [], [⟨K, K, 1⟩])
```

6.6 Final Version of the Program for the MC PROBLEM Derived with KIDS

```
Step 28: Refine
  foci-to-compile: {function MC-TOP (## | ## <= ##) returns(MC-TOP-SET: ## ...
                    , function MC-TOP-GS-AUX-1 (... | ##) returns(## | ##) = ##
                   }
```

function MC-TOP-GS-AUX-1

```
(var ##p attribute false VARIABLE? K: integer,
 var ##p attribute false VARIABLE? V: seq(integer),
 var ##p attribute false VARIABLE? VS
   : seq(tuple(integer, integer, integer)),
 var ##p attribute false VARIABLE? V-SIZE: integer,
 var ##p attribute false VARIABLE? CURRENT-MOVE
   : tuple(integer, integer, integer)
   |
     CURRENT-MOVE = last(VS)
       ∧SEQEQUAL(VS, VISITED-STATES(V, ⟨K, K, 1⟩))
       ∧NO-CANNIBALISM
             (VISITED-STATES(V, ⟨K, K, 1⟩), K)
       ∧POSSIBLE-MOVE(VISITED-STATES(V, ⟨K, K, 1⟩), K)
       ∧NO-MOVES-BACK(VISITED-STATES(V, ⟨K, K, 1⟩))
       ∧ALTERNATE-MOVE(V)
       ∧range(V) ⊆{1 ..10}∧K ≥1
       ∧V-SIZE = size(V))
returns
  (MC-TOP-SET: set(seq(integer)) |
     MC-TOP-SET
       = {MC | (MC: seq(integer))
           EXTENDS(MC, V)
             ∧EVERYBODY-MADE-IT
                   (VISITED-STATES(MC, ⟨K, K, 1⟩))
             ∧NO-MOVES-BACK(VISITED-STATES(MC, ⟨K, K, 1⟩))
             ∧NO-CANNIBALISM
                   (VISITED-STATES(MC, ⟨K, K, 1⟩), K)
             ∧POSSIBLE-MOVE
                   (VISITED-STATES(MC, ⟨K, K, 1⟩), K)
             ∧ALTERNATE-MOVE(MC)
             ∧range(MC) ⊆{1 ..10}})
= {V | () CURRENT-MOVE = ⟨0, 0, -1⟩}
    ∪!reduce
              (UNION!,
               {MC-TOP-GS-AUX-1
                     (K, append(V, I),
                      VS ++[⟨
                                  CURRENT-MOVE.1 + MAPCL-I,
                                  CURRENT-MOVE.2 + MAPML-I,
                                  CURRENT-MOVE.3 *-1⟩],
                      1 + V-SIZE,
                      ⟨CURRENT-MOVE.1 + MAPCL-I,
                            CURRENT-MOVE.2 + MAPML-I,
                            CURRENT-MOVE.3 *-1⟩)
                |
```

```
                    (MAPML-I: integer, MAPCL-I: integer,
                     CURRENT-STATE: seq(tuple(integer, integer, integer)),
                     I: integer)
                    MAPML-I = MAPML(I)
                     ∧
                     CURRENT-STATE
                       = [⟨
                            CURRENT-MOVE.1 + MAPCL-I,
                            CURRENT-MOVE.2 + MAPML-I,
                            CURRENT-MOVE.3 *-1⟩]
                     ∧I ∈{1 ..10}
                     ∧((1 + V-SIZE) mod 2 ≠0 ∧I <6
                              ∨(1 + V-SIZE) mod 2 = 0 ∧5 <I)
                     ∧NO-MOVES-BACK(CURRENT-STATE)
                     ∧CROSS-NO-MOVES-BACK(VS, CURRENT-STATE)
                     ∧POSSIBLE-MOVE(CURRENT-STATE, K)
                     ∧NO-CANNIBALISM(CURRENT-STATE, K)
                     ∧MAPCL-I = MAPCL(I)})

function MC-TOP
    (var ##p attribute false VARIABLE? K: integer | 1 ≤K)
    returns
      (MC-TOP-SET: set(seq(integer)) |
         MC-TOP-SET
           = {MC | (MC: seq(integer))
                range(MC) ⊆{1 ..10}
                  ∧ALTERNATE-MOVE(MC)
                  ∧POSSIBLE-MOVE
                          (VISITED-STATES(MC, ⟨K, K, 1⟩), K)
                  ∧NO-CANNIBALISM
                          (VISITED-STATES(MC, ⟨K, K, 1⟩), K)
                  ∧NO-MOVES-BACK(VISITED-STATES(MC, ⟨K, K, 1⟩))
                  ∧EVERYBODY-MADE-IT
                          (VISITED-STATES(MC, ⟨K, K, 1⟩))})
   = MC-TOP-GS-AUX-1
       (K, [], [⟨K, K, 1⟩], 0,
        ⟨K, K, 1⟩)
```

```
function MC-TOP-GS-AUX-1
    (var ##p attribute false VARIABLE? K: integer,
     var ##p attribute false VARIABLE? V: seq(integer),
     var ##p attribute false VARIABLE? VS
       : seq(tuple(integer, integer, integer)),
     var ##p attribute false VARIABLE? V-SIZE: integer,
     var ##p attribute false VARIABLE? CURRENT-MOVE
       : tuple(integer, integer, integer)
       |
         CURRENT-MOVE = last(VS)
           ∧SEQEQUAL(VS, VISITED-STATES(V, ⟨K, K, 1⟩))
           ∧NO-CANNIBALISM
                 (VISITED-STATES(V, ⟨K, K, 1⟩), K)
           ∧POSSIBLE-MOVE(VISITED-STATES(V, ⟨K, K, 1⟩), K)
           ∧NO-MOVES-BACK(VISITED-STATES(V, ⟨K, K, 1⟩))
           ∧ALTERNATE-MOVE(V)
           ∧range(V) ⊆{1 ..10}∧K ≥1
           ∧V-SIZE = size(V))
   returns
     (MC-TOP-SET: set(seq(integer)) |
        MC-TOP-SET
          = {MC | (MC: seq(integer))
              EXTENDS(MC, V)
                ∧EVERYBODY-MADE-IT
                      (VISITED-STATES(MC, ⟨K, K, 1⟩))
                ∧NO-MOVES-BACK(VISITED-STATES(MC, ⟨K, K, 1⟩))
                ∧NO-CANNIBALISM
                      (VISITED-STATES(MC, ⟨K, K, 1⟩), K)
                ∧POSSIBLE-MOVE
                      (VISITED-STATES(MC, ⟨K, K, 1⟩), K)
                ∧ALTERNATE-MOVE(MC)
                ∧range(MC) ⊆{1 ..10}})
   = {V | () CURRENT-MOVE = ⟨0, 0, -1⟩}
       U!reduce
                 (UNION!,
                  {MC-TOP-GS-AUX-1
                         (K, append(V, I),
                          VS ++[⟨
                                       CURRENT-MOVE.1 + MAPCL-I,
                                       CURRENT-MOVE.2 + MAPML-I,
                                       CURRENT-MOVE.3 *-1⟩],
                          1 + V-SIZE,
                          ⟨CURRENT-MOVE.1 + MAPCL-I,
                                CURRENT-MOVE.2 + MAPML-I,
                                CURRENT-MOVE.3 *-1⟩)
```

```
                    |
                    (MAPML-I: integer, MAPCL-I: integer,
                     CURRENT-STATE: seq(tuple(integer, integer, integer)),
                     I: integer)
                    MAPML-I = MAPML(I)
                      ∧
                     CURRENT-STATE
                       = [⟨
                            CURRENT-MOVE.1 + MAPCL-I,
                            CURRENT-MOVE.2 + MAPML-I,
                            CURRENT-MOVE.3 *-1⟩]
                      ∧I ∈{1 ..10}
                      ∧((1 + V-SIZE) mod 2 ≠0 ∧I <6
                              ∨(1 + V-SIZE) mod 2 = 0 ∧5 <I)
                      ∧NO-MOVES-BACK(CURRENT-STATE)
                      ∧CROSS-NO-MOVES-BACK(VS, CURRENT-STATE)
                      ∧POSSIBLE-MOVE(CURRENT-STATE, K)
                      ∧NO-CANNIBALISM(CURRENT-STATE, K)
                      ∧MAPCL-I = MAPCL(I)})

function MC-TOP
    (var ##p attribute false VARIABLE? K: integer | 1 ≤K)
    returns
      (MC-TOP-SET: set(seq(integer)) |
         MC-TOP-SET
           = {MC | (MC: seq(integer))
                range(MC) ⊆{1 ..10}
                  ∧ALTERNATE-MOVE(MC)
                  ∧POSSIBLE-MOVE
                          (VISITED-STATES(MC, ⟨K, K, 1⟩), K)
                  ∧NO-CANNIBALISM
                          (VISITED-STATES(MC, ⟨K, K, 1⟩), K)
                  ∧NO-MOVES-BACK(VISITED-STATES(MC, ⟨K, K, 1⟩))
                  ∧EVERYBODY-MADE-IT
                          (VISITED-STATES(MC, ⟨K, K, 1⟩))})
    = MC-TOP-GS-AUX-1
        (K, [], [⟨K, K, 1⟩], 0,
         ⟨K, K, 1⟩)
```

References

[Gomes 94] Carla O. Pedro Gomes. O-Plan2 vs. Sipe-2 - A General Comparison. Technical Report RL-TR-94-96, Rome Laboratory, 1994.

[Smith & Parra 93] Doug Smith and Eduardo Parra. Transfornational Approach To Transportation Scheduling. In *Proceedings of the Eigth Knowledge-Based Software Engineering Conference*, Chicago, Illinois, 1993.

[Smith 82] Douglas R. Smith. Derived Preconditions and Their Use in Program Synthesis. In D. W. Loveland, editor, *Sixth Conference on Automated Deduction. Lecture Notes in Computer Science*, volume 138. Berlin: Spring-Verlag, 1982.

[Smith 87] Douglas R. Smith. Structure and Design of Global Search Algorithms. Technical Report KES.U.87.11, Kestrel Institute, 1987.

[Smith 90] Douglas R. Smith. KIDS: A Semi-automated Program Development System. *IEEE Transactions on Software Engineering, Special Issue on Formal Methods*, 16(9):1024–1043, September 1990.

[Smith 91] Douglas R. Smith. KIDS: A Knowledge-based Software Development System . In M. Lowry and R. McCartney, editors, *Automating Software Design*, pages 483–514. MIT Press, 1991.

[Smith 92] Doug Smith. Constructing specification morphisms. Technical Report Tech. Rep. KES.U.92.1, Kestrel Institute, January 1992.

*U.S. GOVERNMENT PRINTING OFFICE: 1995-710-126-20065

MISSION

OF

ROME LABORATORY

Mission. The mission of Rome Laboratory is to advance the science and technologies of command, control, communications and intelligence and to transition them into systems to meet customer needs. To achieve this, Rome Lab:

a. Conducts vigorous research, development and test programs in all applicable technologies;

b. Transitions technology to current and future systems to improve operational capability, readiness, and supportability;

c. Provides a full range of technical support to Air Force Materiel Command product centers and other Air Force organizations;

d. Promotes transfer of technology to the private sector;

e. Maintains leading edge technological expertise in the areas of surveillance, communications, command and control, intelligence, reliability science, electro-magnetic technology, photonics, signal processing, and computational science.

The thrust areas of technical competence include: Surveillance, Communications, Command and Control, Intelligence, Signal Processing, Computer Science and Technology, Electromagnetic Technology, Photonics and Reliability Sciences.