A CLIPS Case Study
A Solution for the Blocks-World Problem

The blocks-world problem is often taken as a basis of study for different general problem solving strategies in AI. Here it is used to exemplify the programming elements of CLIPS and to suggest the extent in which these elements can contribute to a program that reflects clearly and concisely the solving strategy in a way that, moreover, is close to the solved problem.

The problem

The problem has many variants. Here the simplest is chosen. Increasing the problem complexity and modifying the program that solves the simple variant can be taken as an exercise.

Two worlds, called **`Current`** and **`Model`**, are populated with parallelepiped-shaped blocks. The blocks from the two worlds are the same, although they can be arranged differently. The blocks of a given world are arranged according to the following rules:

- There is a unique block, called **`table`** that cannot stand on top of any other block. However, any number of blocks can be placed directly on **`table`**.
- A block different than **`table`** can take a single different block on top of itself.
- Each block, except the **`table`**, must stand on top of another block from that world.

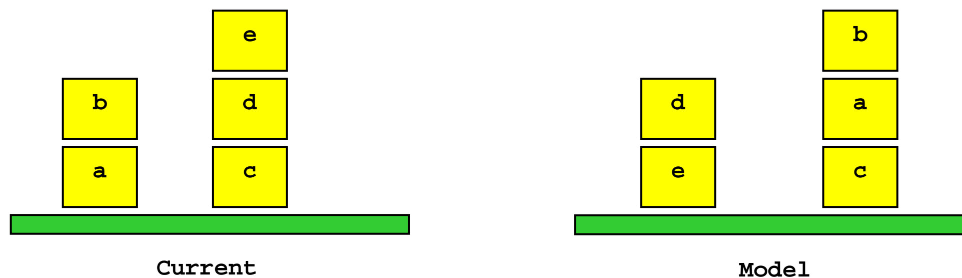

Figure 1. An instance of the blocks world problem

Moving blocks can alter the configuration of a world. Moving a block is subject to the following restrictions:

1. Only a *free* block can be moved from the top of a block onto the top of another *free* block. A block is *free* if it does not support another block (its top is free).
2. The **`table`** cannot be moved.
3. A single block can be moved at a time. There are no concurrent moves.

The goal of the problem is to find a plan of moves that will transform the world **`Current`** according to the configuration of the world **`Model`**, i.e., once the plan is carried out **`Current`** will look alike **`Model`**. For example, the **`Current`** world from figure 1 can be reconfigured to look like the **`Model`**, by following the plan:

```
move e on table
move d on e
move b on table
move a on c
move b on a
```

It is easy to see that an infinity of plans exists for any world with more than two blocks, **table** included. Some plans can be of arbitrary length such as:

```
move e on table
move d on e
move b on table
move b on a
move b on table
move b on a
...
move b on table
move a on c
move b on a
```

There is always a set of optimal plans that have a minimal number of moves. The 5 moves plan above is the optimal plan for the problem in figure 1. In addition, there is always a trivial plan: all blocks are first moved onto the **table** and then to the assigned **Model** positions. This plan requires at most **2(N-2)** moves for a problem with **N** blocks, including the **table**.

When solving the problem the ideal is to find the optimal plan. This is a difficult task, beyond the purpose of this lecture. What is searched for here is a plan better that the trivial plan, and that avoids useless moves of the kind shown above when **b** is repeatedly taken from **a** just to be immediately moved back. In addition, we consider that the configurations of the two worlds are valid and the problem can be solved. Nevertheless, the solution adopted could cope with invalid world configurations. The solving process can stop once the number of moves of the trivial plan is reached thus labeling the problem as non-solvable.

Representing the problem universe

The current state of the problem is represented by facts such as **(on *X Y*)** and **(on_model *X Y*)**, where ***X*** and ***Y*** designate generic blocks.

- The fact **(on *X Y*)** shows that ***X*** is on top of ***Y*** in the world **Current**.
- The fact **(on_model *X Y*)** shows that ***X*** is on top of ***Y*** in the world **Model**.

For each block ***X***, different than **table**, there is a pair of facts **(on *X Y*)** and **(on_model *X Y*)**. The problem in figure 1 is described by the following facts:

```
(on a table)  (on b a)       (on c table)       (on d c)      (on e d)
(on_model a c)(on_model b a) (on_model c table) (on_model d e)(on_model e table)
```

Besides the block relations **on** and **on_model** there is additional factual knowledge used for solving the problem. That knowledge designates *goals* and *differences*.

A *goal* is a particular block relationship to achieve in the **Current** world. A goal is represented by a fact of the form **(goal (block *X*) (on *Y*))** stating that the block ***X*** should stand on top of ***Y*** and, therefore, it should be moved there.

A *difference* is represented by a fact **(difference (block *X*) (on *Y*) (on_model Z))** stating that in the **Current** world the block ***X*** is on top of ***Y***, while in the **Model** it is positioned on top of **Z**.

Goals and differences are not part of the initial problem state. They are automatically generated as the solving process unfolds. It is easy to imagine that goals are generated as effect of differences between the world **Current** and the world **Model**.

Solving the problem

The *means-ends* analysis strategy is used to solve the problem. It is a general problem solving strategy based on the following algorithm:

1. Compare the current state of the problem against the final state and select differences.
2. Chose the most relevant difference and modify the current state of the problem to eliminate the selected difference.
3. Repeat the above two steps until there are no differences between the current and the final problem states.

In our case a difference means a different position of a block in the **Current** world versus the **Model** world. What is meant by relevance is debatable. It is exactly this element that bears direct influence on the quality of the solution. Here, we accept that a difference is relevant if:

- it can be eliminated without extensive changes in the **Current** world and
- once eliminated it does not re-install former differences or block configurations processed during the solving process.

For example, considering the problem in figure 1, take the current solving state:

```
Current= {(on a table) (on b table) (on c table) (on d e) (on e table)}
```

The difference **(difference (block a) (on table) (on_model c))** is a relevant difference, while the difference **(difference (block b) (on table) (on_model a))** is not relevant. Removing this difference would lead back to the same configuration the blocks **a**, **b** and **table** had in the initial state of the problem.

In order to spot relevant differences and to reject non-relevant ones, a new property is defined for each block in the problem. It is said that a block is *settled* if *itself and its supporting blocks*, starting from its direct support down to **table**, are positioned in the **Current** world precisely as they are in the **Model**. This property can be represented by two opposite facts: **(settled *X*)** and, respectively, **(unsettled *X*)**. Since it is easier to determine the *unsettled* property of a block, the solving process uses only facts of the form **(unsettled *X*)**. A block **X** is settled if the fact **(unsettled X)** is not present in the factual knowledge base of the problem.

For the initial problem state in figure 1, the block **c** is settled, while **a**, **b**, **d** and **e** are unsettled. Take the block **b**: indeed its direct support in **Current** is **a**, as in the **Model**, but **a** is not positioned as in the **Model**. Therefore, the block **a** is unsettled and as a direct consequence, **b** is unsettled. In general, seen as a function,

```
                               | false if X = table
unsettled(X,Current,Model) =   |
                               | difference(X,Current,Model) or
                               | unsettled(support(X,Current),Current,Model)
                                       if X ≠ table
```

Translated in terms of facts, the above function is:

- For a fact **(on *X* *Y*)** from **Current**, the fact **(unsettled *X*)** is present in the knowledge base of the problem if and only if:
    - There is a fact **(difference (block *X*)...) or**
    - There is a fact **(unsettled *Y*)**.

- The fact **(unsettled table)** is ruled out automatically, since there is no difference in the positioning of **table**.

A difference **(difference (block *X*) (on *Y*) (on_model *Z*))** is relevant if ***Z*** is settled, i.e. the fact **(unsettled *Z*)** is not present in the knowledge base of the problem. It would be nonsense to move ***X*** on ***Z*** provided ***Z*** has to be moved in the future thus moving again the block ***X***.

Two more decisions have to be taken in order to build the complete solution framework. How a relevant difference is effectively eliminated and how a goal is fulfilled. The difference **Diff(*X*,*Y*,*Z*)=(difference (block *X*) (on *Y*) (on_model *Z*))** is eliminated simply by:

- Generating a goal **(goal (block *X*) (on *Z*))** for moving ***X*** on ***Z***;

- Retracting the fact **Diff(*X*,*Y*,*Z*)** from the factual knowledge base of the program once *there is no evidence to support* the difference. Briefly, the evidence that supports the difference **Diff(*X*,*Y*,*Z*)** is based on the fact **(on *X* *Y*)**. Once this fact is no longer in the factual base of the program, the difference must be retracted. In turn, the fact **(on *X* *Y*)** is retracted when the goal **(goal (block *X*) (on *Z*))** is fulfilled, i.e. when ***X*** is effectively moved and no longer stays on ***Y***.

As far as the fulfillment of a goal **(goal (block *X*) (on *Z*))** is concerned, the following rules apply:

- If ***X*** is free and ***Z*** is free, then the goal is carried out straight away, by effectively moving ***X*** on ***Z***, and by registering the move into the overall plan. Notice that, once the goal is satisfied, ***Z*** is no longer free, and the former support of ***X*** becomes free in the resulting state of the problem.

- If ***X*** is not free and/or ***Z*** is not free, then the goal **(goal (block *X*) (on *Z*))** generates new auxiliary goals for placing on **table** the obstructing blocks that are on top of ***X*** and ***Z***. The goal will be fulfilled only after the auxiliary goals are done with.

Two more observations are worth following:

1. Once fulfilled, the goal **(goal (block *X*) (on *Z*))** is automatically retracted from the factual knowledge base of the problem. It is automatically retracted since there is no evidence to support it. By evidence, we mean the fact **(difference (block *X*) (on *Y*) (on_model *Z*)**.

2. Generating an auxiliary goal **(goal (block *W*) (on table))** to move a block on **table** just to free another block may lead to a recursive block freeing process. For instance, assume that for the **Current** world illustrated in figure 1 the difference to be eliminated is **Diff(a,table,c)= (difference (block a) (on table) (on_model c))**. Then the auxiliary goal **(goal (block d) (on table))** for moving **d** on **table** will lead to the generation of another goal **(goal (block e) (on table))** for moving **e** on **table**. It is also interesting to note that this last auxiliary goal will solve part of the problem, i.e. placing **e** on **table** as in the **Model**. Therefore, problem goals can be solved simply as prerequisites of other goals.

Problem consistency

The above sketch of the solving process makes it clear that there are obvious cause-effect dependencies in the solving universe of the problem. All these dependencies are specified in the CLIPS program using **logical** patterns. They build logical support for effect-facts so that these facts are retracted automatically if there is no logical cause (evidence) to support them.

These problem cause-effect dependencies are illustrated in figure 2. An arrow goes from cause to effect. The figure shows that:

The difference `Diff(X,Y,Z)`, is generated by the presence of the fact `(on X Y)`. Therefore, once the fact `(on X Y)` is retracted from the factual base of the problem, the difference is automatically retracted as well.

The property `(unsettled X)` based on evidence that `X` stays on a block, say `Y`. Once the fact `(on X Y)` is retracted, and replaced by another fact `(on X Z)`, the fact `(unsettled X)` is automatically retracted.

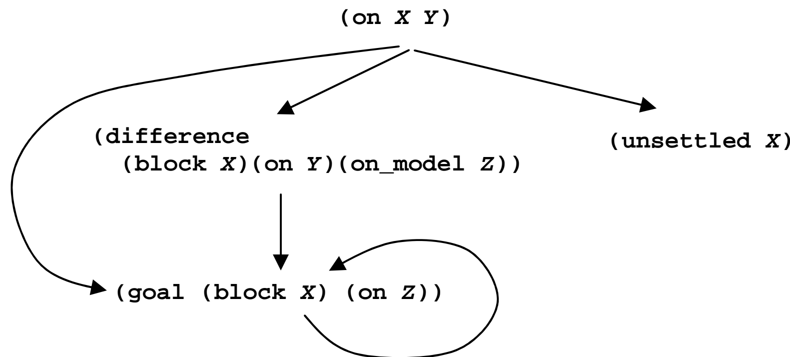

Figure 2. The cause-effect dependency graph for the blocks-world problem

The goal `(goal (block X) (on Z))`, is generated due to a difference `Diff(X,Y,Z)`. Once the difference is no longer present in the factual base of the problem, the goal is automatically retracted.

An auxiliary block-freeing goal `G(W,table)=(goal (block W) (on table))` has an additional cause. There is another parent goal `G'(X,Z)=(goal (block X) (on Z))` that, for being fulfilled, has to free, say `X`, if `X` is busy[1]. The goal `G(W,table)` exists only if a fact `(on W X)` exists. Therefore, `G(W,table)` is automatically retracted when the fact `(on W X)` is retracted. Nevertheless, in the CLIPS program, `G(W,table)` is governed by both the parent goal `G'(X,Z)` and the relation `(on W X)`, although the parent goal has no chance for being fulfilled before the block `X` changes its status. The double dependence is used just to show that cyclic dependencies can be specified in CLIPS.

The program

The program uses the following three modules.

`MAIN` is the repository of the complete factual database. It exports all local constructs and, therefore, the factual base is accessible to the other program modules. The role of `MAIN` is to load the initial state of the problem from a file and to activate the module `SOLVE` as long as there are differences between the `Current` world and the `Model` world.

`SOLVE` solves the existing goals. There can be several goals at any given moment in time, during the solving process of the problem.

[1] There is a similar situation if `Z` is busy. Different block-freeing goals are generated for the block that stands on `X` and for the block that stands on `Z`.

`DIFFERENCE` finds differences that exist between the current `Current` world and the `Model` world. This module is not called explicitly. It activates itself automatically when a modification in the position of a block occurs.

It is interesting to note again the high degree of program non-determinacy. If there is no predefined order to process differences and goals, then the generated plan could be different from program run to program run for the same problem. Nevertheless, the plan will be always correct.

```
; ______________________________________________________________
; Module MAIN
; Load the initial configuration of Current world and the Model
; Keep solving the problem as long as there are differences
; between Current and Model
; ______________________________________________________________

(defmodule MAIN
  (export deftemplate ?ALL))

(deftemplate difference (slot block) (slot on) (slot on_model))
(deftemplate goal (slot block) (slot on))
(deffacts ff (on) (on_model))

(defrule start =>
  (printout t "Input file: ")
  (load-facts (read))
  (set-strategy random))

(defrule solve
  (exists (difference))
  =>
  (refresh solve)
  (focus SOLVE))

; ______________________________________________________________
; Demon DIFFERENCE
; Find all differences between the Current and the Model worlds
; Generate goals for relevant differences
; ______________________________________________________________

(defmodule DIFFERENCE
  (import MAIN deftemplate ?ALL))

(defrule difference
  (declare (auto-focus TRUE))
  (logical (on ?A ?C))
  (on_model ?A ?B&~?C)
  =>
  (assert (difference (block ?A) (on ?C) (on_model ?B))))

(defrule unsettled_block
  (logical (on ?A ?B))
  (or (difference (block ?A)) (unsettled ?B))
  =>
  (assert (unsettled ?A)))

(defrule gen_goal
  (declare (salience -10) (auto-focus TRUE))
  (logical (difference (block ?A) (on ?) (on_model ?B)))
  (not (unsettled ?B))
  =>
  (assert (goal (block ?A) (on ?B))))
```

```
; ___________________________________________________________________
; Module SOLVE
; Solve goals. New goals may be generated to free blocks
; ___________________________________________________________________

(defmodule SOLVE
  (import MAIN deftemplate ?ALL))

(defrule move-block
  (declare (salience 10))
  (goal (block ?A) (on ?B))
  (not (on ? ?A))
  (not (on ? ?B&~table))
  ?relA <- (on ?A ?X)
  =>
  (retract ?relA)
  (assert (on ?A ?B))
  (printout t "move " ?A " from " ?X " on " ?B crlf))

(defrule free-upper
  (logical (goal (block ?A) (on ?))
           (on ?X ?A))
  =>
  (assert (goal (block ?X) (on table))))

(defrule free-lower
  (logical (goal (block ?) (on ?B&~table))
           (on ?X ?B))
  =>
  (assert (goal (block ?X) (on table))))
```

The program is not protected against invalid block relationships. For instance, the program does not terminate if:

```
(on a table)            (on_model a table)
(on b a)                (on_model c a)
(on c b)                (on_model b a)
```

The plan is non-finite since both blocks `b` and `c` are determined to fulfil their conflicting goals and there is nothing to stop them acting foolishly.

```
move c from b on table
move b from a on table
move b from table on a
move b from a on table
move c from table on a
move c from a on table
move c from table on a
move c from a on table
move b from table on a . . .
```

Similarly, the program will loop forever if the problem is

```
(on a table)            (on_model a b)
(on b a)                (on_model b a)
```

There is a permanent fact `(difference (block a) (on table) (on_model b))` in the factual knowledge base of `MAIN` and the rule `solve` will continue firing forever. However, this time the dream of block `a` cannot materialize since the block `b` is unsettled, although `b` stays as specified in the model. There will be no moves, but an endless dream of an unfulfilled ideal for `a`, and the nightmare of an unsteady happiness for `b`.

Apart from modeling psychotic cases as above the program works fine. A trace for the problem in figure 1 shows exactly the way the solving process unfolds.

```
<== Focus MAIN
==> Focus MAIN
==> f-0     (initial-fact)
==> f-1     (on)
==> f-2     (on_model)
==> f-3     (unsettled)
CLIPS> (run)
Input file: h:\Cpsc-432\CLIPSprg\Blocks1.dat
```

```
==> f-4     (on a table)                              (load-facts (read))
==> f-5     (on b a)
==> f-6     (on c table)
==> f-7     (on d c)
==> f-8     (on e d)
==> f-9     (on_model a c)
==> Focus DIFFERENCE from MAIN
==> f-10    (on_model b a)
==> f-11    (on_model c table)
==> f-12    (on_model d e)
==> f-13    (on_model e table)
```

DIFFERENCE is not active until the current rule terminates execution.

```
==> f-14    (difference (block a) (on table) (on_model c))
==> f-15    (difference (block d) (on c) (on_model e))
==> f-16    (unsettled d)
==> f-17    (unsettled e)
==> f-18    (unsettled a)
==> f-19    (unsettled b)
==> f-20    (difference (block e) (on d) (on_model table))
==> f-21    (goal (block a) (on c))
==> f-22    (goal (block e) (on table))

<== Focus DIFFERENCE to MAIN
==> Focus SOLVE from MAIN
```

```
<== f-8     (on e d)                                  move e from d on table
<== f-17    (unsettled e)
==> Focus DIFFERENCE from SOLVE
<== f-20    (difference (block e) (on d) (on_model table))
<== f-22    (goal (block e) (on table))
==> f-23    (on e table)
move e from d on table
```

Block d can move since e is settled.

```
==> f-24    (goal (block d) (on e))
<== Focus DIFFERENCE to SOLVE
```

```
<== f-7     (on d c)                                  move d from c on e
<== f-15    (difference (block d) (on c) (on_model e))
<== f-24    (goal (block d) (on e))
<== f-16    (unsettled d)
==> f-25    (on d e)
move d from c on e
```

```
==> f-26    (goal (block b) (on table))          move  b on table
<== f-5     (on b a)                           since a moves on c
<== f-19    (unsettled b)
<== f-26    (goal (block b) (on table))
==> f-27    (on b table)
==> Focus DIFFERENCE from SOLVE
move b from a on table
```

Block b must wait since a is unsettled.

```
==> f-28    (difference (block b) (on table) (on_model a))
==> f-29    (unsettled b)
<== Focus DIFFERENCE to SOLVE
```

```
                                          move a from table on c
<== f-4     (on a table)
<== f-14    (difference (block a) (on table) (on_model c))
<== f-21    (goal (block a) (on c))
<== f-18    (unsettled a)
==> Focus DIFFERENCE from SOLVE
==> f-30    (on a c)
move a from table on c
```

Block b can move since a is settled.

```
==> f-31    (goal (block b) (on a))
<== Focus DIFFERENCE to SOLVE
```

```
                                          move b from table on a
<== f-27    (on b table)
<== f-28    (difference (block b) (on table) (on_model a))
<== f-31    (goal (block b) (on a))
<== f-29    (unsettled b)
==> f-32    (on b a)
move b from table on a
```

```
<== Focus SOLVE to MAIN
<== Focus MAIN
```

Making the program robust

As an exercise let us perform the minimal modifications to the above program such that it can work with invalid world configurations. The modified program will have only one new rule, which just warns for invalid world configurations. We have to observe the following:

1. The solving process should stop when there are no more goals. Indeed the goals are created only for relevant differences. Therefore, differences for faulty **`Model`** configurations of the kind **`(on_model a b)`**, **`(on_model b a)`** will not create goals and the program will stop. This observation leads to a slightly modified rule **`solve`** from **`MAIN`**. The rule applies only when there **`(exists (goal))`** instead of **`(exists (difference))`**.

2. If there is no goal but there are still differences, it means that all differences are not relevant and, therefore, the problem is faulty. There is a new rule **`faulty_problem`** in the module **`MAIN`** that watches for such an event. When fired it will issue a warning. Practically, this rule is terminal

since, after it fires, the full program will be blocked even if the **Agenda** of the **DIFFERENCE** module is not empty.

3. The program must refuse to solve the same goal twice. To account for already solved goals a new template is declared

```
(deftemplate solved_goal (slot block) (slot on))
```

The fact **(solved_goal (block *X*) (on *Y*))** is generated whenever the rule **move-block** from the module **SOLVE** effectively solves the goal **(goal (block *X*) (on *Y*))**. In addition, the generation of a new goal is conditioned by the absence of the same **solved_goal** fact. The rule **gen_goal** from the module **DIFFERENCE** gets the additional pattern **(not (solved_goal ...))**. For example, in the case

```
(on a table)          (on_model a table)
(on b a)              (on_model c a)
(on c b)              (on_model b a)
```

once solved, the goals **(goal (block b) (on a))** and **(goal (block c) (on a))** will not be generated again. In this way endless loops are avoided.

The modified program terminates even in the faulty cases mentioned above, when the initial program loops forever.

```
; _______________________________________________________________
; Module MAIN
; Load the initial configuration of Current world and the Model
; Keep solving the problem as long as there are goals to solve
; Stop with:
;         a) success if there are no goals and no differences
;         b) warning for faulty worlds if there are differences
;            but no goals to solve
; _______________________________________________________________

(defmodule MAIN
  (export deftemplate ?ALL))

(deftemplate goal (slot block) (slot on))
(deftemplate solved_goal (slot block) (slot on))
(deftemplate difference (slot block) (slot on) (slot on_model))
(deffacts ff (on) (on_model))

(defrule start =>
  (printout t "Input file: ")
  (load-facts (read))
  (set-strategy random))

(defrule solve
  (exists (goal))
  =>
  (refresh solve)
  (focus SOLVE))

(defrule faulty_problem
  (not (goal))
  (exists (difference))
  =>
  (printout t crlf "Invalid problem specification"
            crlf "The plan (if any) is not complete" crlf))
```

```
; _____________________________________________________________
; Demon DIFFERENCE
; Find all differences between the Current and the Model worlds
; Generate goals for relevant differences. Avoid endless loops
; by keeping track of solved goals
; _____________________________________________________________

(defmodule DIFFERENCE
  (import MAIN deftemplate ?ALL))

(defrule difference
  (declare (auto-focus TRUE))
  (logical (on ?A ?C))
  (on_model ?A ?B&~?C)
  =>
  (assert (difference (block ?A) (on ?C) (on_model ?B))))

(defrule unsettled_block
  (logical (on ?A ?B))
  (or (difference (block ?A)) (unsettled ?B))
  =>
  (assert (unsettled ?A)))

(defrule gen_goal
  (declare (salience -10) (auto-focus TRUE))
  (logical (difference (block ?A) (on ?) (on_model ?B)))
  (not (unsettled ?B))
  (not (solved_goal (block ?A) (on ?B)))
  =>
  (assert (goal (block ?A) (on ?B))))

; ____________________________________________________________
; Module SOLVE
; Solve goals. New goals may be generated to free blocks
; ____________________________________________________________

(defmodule SOLVE
  (import MAIN deftemplate ?ALL))

(defrule move-block
  (declare (salience 10))
  (goal (block ?A) (on ?B))
  (not (on ? ?A))
  (not (on ? ?B&~table))
  ?relA <- (on ?A ?X)
  =>
  (retract ?relA)
  (assert (on ?A ?B)
          (solved_goal (block ?A) (on ?B)))
  (printout t "move " ?A " from " ?X " on " ?B crlf))

(defrule free-upper
  (logical (goal (block ?A) (on ?))
           (on ?X ?A))
  =>
  (assert (goal (block ?X) (on table))))

(defrule free-lower
  (logical (goal (block ?) (on ?B&~table))
           (on ?X ?B))
  =>
  (assert (goal (block ?X) (on table))))
```

The easy way of mending the program is remarkable. Compare it with the case of an imperative program where new clumsy code should be added to explicitly test for faulty cases. The declarative programming style pays off too in this respect. Moreover, the new program has an interesting characteristic. If the problem is only partly correct, i.e. there are invalid block relationships, the program solves as much as it could be solved, leaving the faulty part of the problem unsolved or partially solved. For instance, consider the following problem, partly faulty:

Current	Model
(on b table)	(on_model b x)
(on a b)	(on_model b a)
(on c a)	(on_model d table)
(on d c)	(on_model c d)
(on f table)	(on_model a c)
(on table g)	(on_model e f)
(on e f)	(on_model f e)

The `table` stays on a block from `Current` (error). The block `b` must stand on two blocks (error) out of which `x` is unknown (error). In addition, `e` and `f` must stand each one on the other one (error). The single valid part from the problem is the tower of blocks `(on_model d table)`, `(on_model c d)`, `(on_model a c)`. The program synthesizes the following plan:

move d from c on table move c from a on d move a from b on c	Correct plan fragment for the correct part of the problem
move b from table on a move b from a on x Invalid problem specification The plan (if any) is not complete	Partly correct plan fragment. The block `b` cannot stand simultaneously on two blocks. Finally, it is placed on `x`, although `x` does not exist.

The plan contains no moves for the blocks `e` and `f`. This part of the problem cannot be solved at all.