

Deduction-based Refinement Planning*

Werner Stephan Susanne Biundo

German Research Center for Artificial Intelligence (DFKI)
Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany
e-mail: {stephan,biundo}@dfki.uni-sb.de

Abstract

We introduce a method of deduction-based refinement planning where pre-fabricated general solutions are adapted to special problems. Refinement proceeds by stepwise transforming non-constructive problem specifications into executable plans. For each refinement step there is a correctness proof guaranteeing the soundness of refinement and with that the generation of provably correct plans. By solving the hard deduction problems once and for all on the abstract level, planning on the concrete level becomes more efficient. With that, our approach aims at making deductive planning feasible in realistic contexts.

Our approach is based on a temporal logic framework that allows for the representation of specifications and plans on the same linguistic level. Basic actions and plans are specified using a programming language the constructs of which are formulae of the logic. Abstract solutions are represented as—possibly recursive—procedures. It is this common level of representation and the fluid transition between specifications and plans our refinement process basically relies upon.

*DFKI Research Report RR-95-13, 1995. Also in: Proceedings of the 3rd International Conference on Artificial Intelligence Planning Systems (AIPS-96).

Contents

1	Introduction	3
2	The Logic	4
3	Planning Scenarios and Planning Problems	6
4	Abstract Solutions	8
5	Refinement Planning	9
6	Related Work and Conclusion	13

1 Introduction

In this paper, we present a technique for deduction-based refinement planning. The idea is to refine an initial non-constructive specification step by step until an executable plan is reached. Since each refinement step is sound under the proviso of certain proof obligations, we end up with provably correct plans.

The method is developed within a temporal logic framework similar to [MP91, SB93]. Specifications and plans are represented on the same linguistic level and they both specify sequences of states, called computations. As a consequence, we are not limited to the input-output behavior of plans when formulating specifications. Instead, we may also state properties of certain intermediate states and in particular so-called safety conditions which have to hold in all intermediate states.

The refinement process takes a non-constructive initial specification and generates an executable one, i.e. a plan, the set of computations of which is included in the set of computations described by the initial specification.

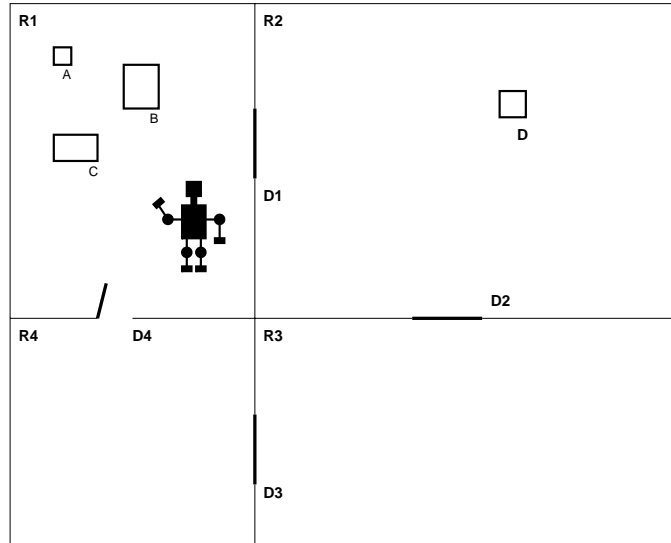


Figure 1: The 4-Room Planning Scenario

Our aim is to make deductive planning feasible in realistic contexts. Therefore, we follow the paradigm of hierarchical planning. The hard deduction problems, like proving the total correctness of recursive plans, are solved once and for all on an upper level by providing a collection of prefabricated abstract and general plans. Plan generation on this level is an interactive process with non-trivial inferences that in our opinion, which is shared by other authors as well (cf. [GSS91, MW87]), cannot be carried out in a fully automatic way. These abstract algorithmic solutions are in an efficient way refined to specific concrete ones.

Consider, for example, the concrete planning scenario of Figure 1 and suppose we are given the problem of moving blocks A, B, and C from R1 to R4. Starting from scratch, it is very likely that the planning process gets lost in a mass of subproblems such as to find the right door for B. It would be of considerable help for the planner to know that the solution should be organized in three phases, where in each phase one block is moved from R1 to R4. In our approach this additional knowledge is given as an abstract plan with certain open steps that are filled in by concrete plans during later refinement. Not only is this a way to formalize additional knowledge, it also reduces the necessary proofs to the local intermediate steps. The search for appropriate steps on the lower level is done by matching concrete actions to elementary state transitions given by the abstract solution. Abstraction here means that we do not consider the world in all its details but restrict ourselves to certain features of a planning scenario that are sufficient to outline a solution. Since general solutions often require the manipulation of an indefinite number of objects, plans at this level will in many cases have to be *recursive*. In our approach, recursive plans and the corresponding correctness proofs are part of the domain modeling on the abstract level. A further important point is that we provide a uniform algorithmic solution only on the abstract level while the final plan is sensitive to additional ad hoc constraints for which there is no uniform treatment.

We first introduce our basic representation formalism and then show—by means of a detailed example—how refinement planning works in this context.

2 The Logic

We use an interval-based modal temporal logic to formally reason about plans. The syntax of a planning scenario is given by a so-called *language* $\mathcal{L} = (Z, \mathbf{F}_r, \mathbf{R}_r, \mathbf{F}_f, \mathbf{R}_f, \mathbf{X}, \mathbf{A})$, where Z is a finite set of *sort symbols*, \mathbf{F}_f and \mathbf{F}_r are disjoint $Z \times Z^*$ indexed families of disjoint sets of *function symbols*, \mathbf{R}_f and \mathbf{R}_r are disjoint Z^* indexed families of disjoint sets of *relation symbols*, \mathbf{X} is a Z indexed family of disjoint denumerable sets of global *variables*, such that $X_z \cap F_{z,\varepsilon} = \{\}$ for all $z \in Z$, and \mathbf{A} is a Z^* indexed family of disjoint denumerable sets of *abstraction* symbols (procedure names) such that $A_{\bar{z}} \cap (R_{f,\bar{z}} \cup R_{r,\bar{z}}) = \{\}$ for all $\bar{z} \in Z^*$. Abstraction symbols will be used to describe basic actions. The *rigid* and *flexible* symbols are given by $(\mathbf{F}_r, \mathbf{R}_r)$ and $(\mathbf{F}_f, \mathbf{R}_f)$, respectively. The flexible symbols will be interpreted in a state dependent way. In the following, a simplified version will be considered where there are no flexible function symbols.

Rigid *terms* t over \mathcal{L} containing symbols from \mathbf{F}_r and \mathbf{X} are built as usual. The set of *formulae* and *abstractions* over \mathcal{L} is given by the following rules:

$$\begin{aligned} \varphi & ::= t_1 \equiv t_2 \mid r(\bar{t}) \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \forall x \varphi \mid \text{O}\varphi \mid \varphi_1 \mathcal{U} \varphi_2 \mid \\ & \quad \varphi_1 \mathcal{C} \varphi_2 \mid \gamma(\bar{t}) \mid \mathbf{delete}\text{-}r(\bar{t}) \mid \mathbf{add}\text{-}r(\bar{t}) \\ \gamma & ::= a \mid \lambda \bar{x}.\varphi \mid \nu a(\bar{x}).\varphi . \end{aligned}$$

\circ and \mathcal{U} denote the modal *weak next* and *until* operators, respectively. \mathcal{C} denotes the *chop* operator, which serves to express the sequential composition of formulae. The relation symbols used in the context of **delete** and **add** have to be flexible. Using the until operator \mathcal{U} we define the usual modalities \Box (*always*) and \Diamond (*sometimes*) by $\Box\varphi :\leftrightarrow \neg(\text{true } \mathcal{U} \neg\varphi)$ and $\Diamond\varphi :\leftrightarrow \neg\Box\neg\varphi$. The semantics of the recursively defined abstraction $\nu a(\bar{x}).\varphi$ will be such that the equation $\nu a(\bar{x}).\varphi = \lambda\bar{x}.\varphi[a/\nu a(\bar{x}).\varphi]$ is satisfied, provided $\varphi[a]$ is syntactically continuous in a . Moreover, under these circumstances $\nu a(\bar{x}).\varphi$ is the maximal solution of the corresponding equation.

A model \mathcal{M} for a language \mathcal{L} is given by a Z -indexed family \mathbf{D} of nonempty *domains* and a global *interpretation* \mathcal{I} that assigns (total) functions and relations over \mathbf{D} to the rigid symbols. A *valuation* (of variables) w.r.t. \mathcal{M} is a sort preserving mapping $\beta : |\mathbf{X}| \rightarrow |\mathbf{D}|$. We use $\beta(x/d)$ for the valuation β' which satisfies $\beta =_x \beta'$ (β and β' agree on all arguments except possibly x) and $\beta'(x) = d$. The set of *states* Σ w.r.t. \mathcal{M} is the set of all interpretations of the flexible relation symbols as relations over \mathbf{D} . Rigid terms are evaluated by using \mathcal{M} and β as usual.

As is the case with *choppy* logics in general, the semantics is based on *intervals* of *states* [RP86]. We use Σ^∞ to denote the set of all finite and infinite sequences $\bar{\sigma} = \langle \sigma_0, \sigma_1, \dots \rangle$ of states from Σ and Σ^{\aleph} for $\Sigma^\infty - \{\langle \rangle\}$. By Σ^i we denote the set of sequences of length i . The concatenation of intervals is denoted by “.”. *Fusion*, denoted by “ \circ ”, is the partial operation on intervals defined by:

$$\bar{\sigma} \circ \bar{\sigma}' = \begin{cases} \bar{\sigma} & \text{if } \bar{\sigma} \text{ is infinite} \\ \langle \sigma_0, \dots, \sigma_n, \dots \rangle & \text{if } \bar{\sigma} = \langle \sigma_0, \dots, \sigma_n \rangle, \bar{\sigma}' = \langle \sigma_n, \dots \rangle \end{cases} .$$

Both, concatenation and fusion are extended to sets of intervals in the usual way. For \mathcal{M} and a valuation β the semantics of formulae is given by $\llbracket \varphi \rrbracket_{\mathcal{M}, \beta} \subseteq \Sigma^{\aleph}$. For atomic formulae ϕ containing only rigid symbols $\llbracket \phi \rrbracket_{\mathcal{M}, \beta}$ is either Σ^{\aleph} or $\{\}$. The interpretation of the propositional connectives is also straightforward. The more interesting cases are as follows.

$$\begin{aligned} \llbracket r(\bar{t}) \rrbracket_{\mathcal{M}, \beta} &= \bigcup \{ \{\sigma\} \cdot \Sigma^\infty \mid \sigma(r) (\llbracket \bar{t} \rrbracket_{\mathcal{M}, \beta}) \}, \text{ for flexible } r \\ \llbracket \forall x.\varphi \rrbracket_{\mathcal{M}, \beta} &= \bigcap \{ \llbracket \varphi \rrbracket_{\mathcal{M}, \beta(x/d)} \mid d \in D_x \}, \text{ for } x \in X_z \\ \llbracket \circ\varphi \rrbracket_{\mathcal{M}, \beta} &= \Sigma^1 \cup (\Sigma^1 \cdot \llbracket \varphi \rrbracket_{\mathcal{M}, \beta}) \\ \llbracket \varphi_1 \mathcal{U} \varphi_2 \rrbracket_{\mathcal{M}, \beta} &= \bigcup \{ A_i \mid i \geq 0 \}, \text{ where} \\ &\quad A_i = \Sigma^i \cdot \llbracket \varphi_2 \rrbracket_{\mathcal{M}, \beta} \cap \{ \bigcap \{ \Sigma^j \cdot \llbracket \varphi_1 \rrbracket_{\mathcal{M}, \beta} \mid 0 \leq j < i \} \} \\ \llbracket \varphi_1 \mathcal{C} \varphi_2 \rrbracket_{\mathcal{M}, \beta} &= \llbracket \varphi_1 \rrbracket_{\mathcal{M}, \beta} \circ \llbracket \varphi_2 \rrbracket_{\mathcal{M}, \beta} \\ \llbracket \gamma(\bar{t}) \rrbracket_{\mathcal{M}, \beta} &= \llbracket \gamma \rrbracket_{\mathcal{M}, \beta} (\llbracket \bar{t} \rrbracket_{\mathcal{M}, \beta}) \\ \llbracket \mathbf{add}\text{-}r(\bar{t}) \rrbracket_{\mathcal{M}, \beta} &= \{ \langle \sigma_0, \sigma_1 \rangle \mid \sigma_1 =_r \sigma_0, \sigma_1(r) = \sigma_0(r) \cup (\llbracket \bar{t} \rrbracket_{\mathcal{M}, \beta}) \} \\ \llbracket \mathbf{delete}\text{-}r(\bar{t}) \rrbracket_{\mathcal{M}, \beta} &= \{ \langle \sigma_0, \sigma_1 \rangle \mid \sigma_1 =_r \sigma_0, \sigma_1(r) = \sigma_0(r) - (\llbracket \bar{t} \rrbracket_{\mathcal{M}, \beta}) \} \end{aligned}$$

For abstractions we have

$$\begin{aligned}
\llbracket \lambda \bar{x}. \varphi \rrbracket_{\mathcal{M}, \beta}(\bar{d}) &= \llbracket \varphi \rrbracket_{\mathcal{M}, \beta(\bar{x}/\bar{d})} \\
\llbracket a \rrbracket_{\mathcal{M}, \beta}(\bar{d}) &= \Sigma^{\mathfrak{A}} \\
\llbracket \nu a(\bar{x}). \varphi \rrbracket_{\mathcal{M}, \beta}(\bar{d}) &= \bigcap \{ \llbracket \gamma_i \rrbracket(\bar{d}) \mid i \geq 0 \}, \text{ where} \\
&\quad \gamma_0 = a \text{ and } \gamma_{i+1} = \lambda \bar{x}. \varphi[a/\gamma_i]
\end{aligned}$$

Note that quantification is over global variables and that first-order formulae are evaluated in the first state of an interval.

The logic presented above can be used to describe computations of certain formulae (plans) that can be viewed as programming language constructs. Elementary operations are **add- r** and **delete- r** for each flexible relation symbol r , while basic actions are described as procedural abstractions. In order to ease readability we use the following abbreviations:

$$\begin{aligned}
\mathbf{if } \phi \mathbf{ then } \pi_1 \mathbf{ else } \pi_2 \mathbf{ fi} &:\leftrightarrow (\phi \rightarrow \pi_1) \wedge (\neg \phi \rightarrow \pi_2) \\
\mathbf{choose } \bar{x} : \phi(\bar{x}) \mathbf{ begin } \pi \mathbf{ end} &:\leftrightarrow \exists \bar{x}. (\phi(\bar{x}) \wedge \pi(\bar{x})) \vee (\forall \bar{x}. \neg \phi(\bar{x}) \wedge \mathbf{skip}) \\
\pi_1 ; \pi_2 &:\leftrightarrow \pi_1 \mathcal{C} \pi_2 \\
\mathbf{skip} &:\leftrightarrow \bigcirc \mathbf{false} \\
a(\bar{x}) \Leftarrow \pi &:\leftrightarrow \nu a(\bar{x}). \pi .
\end{aligned}$$

For recursively defined abstractions we have the equivalence $\nu a(\bar{x}). \pi(\bar{t}) \leftrightarrow \pi[\bar{x}/\bar{t}] [a/\nu a(\bar{x}). \pi]$.

3 Planning Scenarios and Planning Problems

Planning scenarios are set up by first giving a set of sort symbols and a signature of flexible and rigid relation symbols. In our example (cf. Figure 1), these are: $Z = \{block, door, room\}$, $R_f = \{closed, holds, in, rob\}$ and $R_r = \{broad, connects, small\}$, respectively. In a second step basic actions are defined by procedures, like

$$\begin{aligned}
\text{pickup}(b) &\Leftarrow \mathbf{if } \exists r (in(b, r) \wedge rob(r)) \wedge \neg \exists x holds(x) \\
&\quad \mathbf{then add-holds}(b) \mathbf{ else skip fi} \\
\text{walk}(r_1, r_2) &\Leftarrow \mathbf{if } rob(r_1) \wedge \exists d (connects(d, r_1, r_2) \wedge \neg closed(d)) \\
&\quad \mathbf{then if } \exists x holds(x) \mathbf{ then choose } x : holds(x) \\
&\quad \quad \mathbf{begin if } small(x) \vee broad(d) \\
&\quad \quad \quad \mathbf{then delete-in}(x, r_1); \mathbf{delete-rob}(r_1); \\
&\quad \quad \quad \quad \mathbf{add-rob}(r_2); \mathbf{add-in}(x, r_2) \\
&\quad \quad \quad \mathbf{else skip fi end} \\
&\quad \quad \mathbf{else delete-rob}(r_1); \mathbf{add-rob}(r_2) \mathbf{ fi} \\
&\quad \mathbf{else skip fi} .
\end{aligned}$$

From these procedures certain sets of formulae are generated. They comprise *action descriptions*, *effect descriptions*, and *invariance clauses* and they serve to perform the various specific tasks which occur during refinement planning. Action descriptions specify the cases in which an action “really acts”, i.e. its body differs from **skip**. For “walk” we obtain two formulae W_1 and W_2 , where:¹

$$\begin{aligned} W_1 : & (rob(r_1) \wedge connects(d, r_1, r_2) \wedge \neg closed(d) \wedge holds(x)) \wedge \\ & (small(x) \vee broad(d)) \rightarrow (walk(r_1, r_2) \leftrightarrow \\ & \mathbf{delete-in}(x, r_1); \mathbf{delete-rob}(r_1); \mathbf{add-rob}(r_2); \mathbf{add-in}(x, r_2)) . \end{aligned}$$

Action descriptions are used to instantiate abstract solution patterns with concrete solutions. Effect descriptions indicate the immediate effects an action has. They closely correspond to the action descriptions. For “pickup”, for example, we obtain only one such axiom:

$$\begin{aligned} P_{eff} : & (\exists r (in(b, r) \wedge rob(r)) \wedge \neg \exists x holds(x) \wedge pickup(b)) \\ & \rightarrow \square (\circ \mathbf{false} \rightarrow holds(b)) . \end{aligned}$$

Invariance clauses specify the facts that are not affected by the action. They comprise *invariance assertions* $\mathbf{inv-r}(\bar{x}) : \phi(\bar{x})$, which stand for formulae

$$\forall \bar{x} ((\phi(\bar{x}) \wedge r(\bar{x})) \rightarrow \circ r(\bar{x})) \wedge \forall \bar{x} ((\phi(\bar{x}) \wedge \neg r(\bar{x})) \rightarrow \circ \neg r(\bar{x})) ,$$

where ϕ is first-order. As for “pickup” we have

$$\begin{aligned} P_{inv} : & pickup(b) \rightarrow \square (\mathbf{inv-in}(b', r) : \mathbf{true} \wedge \mathbf{inv-closed}(d) : \mathbf{true} \wedge \\ & \mathbf{inv-rob}(r) : \mathbf{true}) , \end{aligned}$$

stating that “pickup” doesn’t change any of the relations *in*, *closed*, and *rob*.

The above formulae can be generated in a uniform way by a purely syntactic inspection of the user-defined procedures, and they can be easily proved using, for example, the equivalences for control structures given in Section 2.

A possible format for *initial specifications* φ_0 is

$$\mathbf{FIN} \wedge \mathbf{EF} \wedge \mathbf{SAFE} \wedge \mathbf{INV} .$$

These four conjuncts describe properties a computation has to meet in order to be accepted as a possible solution. We have

$$\begin{aligned} \mathbf{FIN} & : \leftrightarrow \diamond \circ \mathbf{false} , \\ \mathbf{EF} & : \leftrightarrow \phi_{pre} \rightarrow \square (\circ \mathbf{false} \rightarrow \phi_{post}) , \\ \mathbf{SAFE} & : \leftrightarrow \phi_{pre} \rightarrow \square (\phi_{safe_1} \wedge \dots \wedge \phi_{safe_n}) , \text{ and} \\ \mathbf{INV} & : \leftrightarrow \square (\mathbf{inv-r}_1(\bar{x}_1) : \phi_1(\bar{x}_1) \wedge \dots \wedge \mathbf{inv-r}_m(\bar{x}_m) : \phi_m(\bar{x}_m)) . \end{aligned}$$

¹We write $walk(r_1, r_2)$ instead of the complete recursive definition $walk(r_1, r_2) \Leftarrow \pi(r_1, r_2)$.

The formula FIN states that the computations are *finite*. The *effect* of a computation, given by EF, is described by pre- and postconditions which are first-order formulae. In addition to the desired effect the initial specification may contain *safety conditions* which have to hold in all intermediate states. Again, these conditions are first-order. Finally, INV serves to specify the facts that have to remain unchanged. Before starting refinement planning with a concrete problem specification, we have to state the *domain facts*, which are relevant for the current scenario. Domain facts describe those facts which are not affected by actions, i.e. remain static in a current scenario, but which may vary from one concrete scenario to another. These facts are described using the rigid part of the signature. For our example (cf. Figure 1), we obtain:

DF : *connects*(D1, R1, R2) , *connects*(D1, R2, R1) , *connects*(D2, R2, R3) ,
connects(D2, R3, R2) , *connects*(D3, R3, R4) , *connects*(D3, R4, R3) ,
connects(D4, R4, R1) , *connects*(D4, R1, R4) , *small*(A) , *small*(D) ,
 \neg *small*(B) , \neg *small*(C) , *broad*(D1) , *broad*(D2) , *broad*(D3) ,
 \neg *broad*(D4) .

4 Abstract Solutions

Below we are going to describe a strategy where abstract recursive patterns are refined to plans made up of basic actions from a lower, more concrete, level. In our example, the abstract level consists of a scenario where we only have blocks and rooms which are related by *in*. The problem of moving an arbitrary set of blocks from one room to another is then solved by the following recursive plan:

```

move (s, r1, r2)  $\Leftarrow$  if  $\forall b (b \in s \rightarrow in(b, r_1)) \wedge s \neq \emptyset$  then
    choose b : b  $\in$  s begin
        delete-in(b, r1) ; add-in(b, r2) ; move(s - {b}, r1, r2) end else skip fi

```

It moves one block after the other from r_1 to r_2 . Here we have used the *abstract data type* of *sets*. The signature of this data type contains the *rigid* symbols \in , \emptyset , $\{\dots\}$, and $(-)$. The idea now is to refine this plan on a lower level where in addition there are doors of different size and a robot, but where the “move” operation is no longer available. In a sense this abstract operation has to be implemented by sequences of operations from the lower level. In order to enable such a refinement we have to allow for certain additional steps on the abstract level, thereby extending the set of possible computations. To this end, we adopt a method known as *stuttering* [Lam94, MM94]. It allows for the insertion of additional steps which however do not affect the facts we are interested in on the abstract level so that we are still able to prove useful facts about the abstract solution. Stuttering versions of abstract

operations still exhibit essentially the same behavior while they leave room for later refinements.

Stuttering is introduced by replacing the basic add- and delete-operations in the body of abstract plans, “move” in our case, by appropriate stuttering versions. So, **delete-in**(b, r_1), for example, is replaced by

$$\text{STUT}_{\text{move}} ; \mathbf{delete-in}(b, r_1) ; \text{STUT}_{\text{move}} , \quad \text{where } \text{STUT}_{\text{move}} \text{ is the formula}$$

$$\diamond \circ \mathbf{false} \wedge \square \mathbf{inv-in}(b', r) : (b \neq b' \vee r = r_1 \vee r = r_2) .^2$$

This allows for inserting certain steps between the essential add- and delete-operations. However, $\text{STUT}_{\text{move}}$ forces these steps to be safe in the sense that they must not affect the *in* relation for blocks different from the one just manipulated, as well as *in* w.r.t. the current rooms r_1 and r_2 . The specification of abstract plans together with their STUT formulae is subject to the process of domain modeling, like it was sketched in Section 3 for actions and planning scenarios. Replacing the add- and delete-operations in our example by their stuttering versions using $\text{STUT}_{\text{move}}$ in both cases, we obtain the abstract plan $\text{move}^*(s, r_1, r_2)$. We are able to prove the assertions

$$\text{move}^*(s, r_1, r_2) \rightarrow \text{FIN} ,$$

$$\text{move}^*(s, r_1, r_2) \rightarrow \mathbf{inv-in}(b, r) : b \notin s ,$$

$$\text{move}^*(s, r_1, r_2) \rightarrow \forall b (b \in s \rightarrow \mathbf{in}(b, r_1)) \rightarrow$$

$$\square (\circ \mathbf{false} \rightarrow \forall b (b \in s \rightarrow \mathbf{in}(b, r_2))) ,$$

describing termination of the stuttering version of “move”, its invariance properties, and its effects, respectively. Given an initial specification, refinement planning will start from abstract plans of this form and proceed by stepwise filling up the STUT gaps with concrete plans.

5 Refinement Planning

In our environment, refinement planning consists in transforming an initial specification—via intermediate steps—to a concrete plan. For each step there is a correctness proof guaranteeing the soundness of refinement and with that provably correct plans. In the refinement process we try to construct a plan π the set of computations $\hat{\pi}$ of which is a subset of $\hat{\varphi}_0$. This is done by transforming φ_0 gradually by a sequence of intermediate specifications

$$\varphi_0 \succ \varphi_1 \succ \dots \succ \varphi_n = \pi$$

²Note that b, r_1 , and r_2 always correspond to the current arguments of the “move” call.

to a *plan formula* π . In each step, we restrict the set of computations, that is we have $\varphi_{i+1} \rightarrow \varphi_i$, for all $0 \leq i < n$.

Now we are going to describe a particular refinement strategy that adapts a general and abstract solution to a special problem. It proceeds in three phases:

- *Phase 1*: Find a solution on the abstract level.
- *Phase 2*: Unwind recursive plans.
- *Phase 3*: Fill in missing steps.

We will begin with an initial problem specification which has the form of φ_0 given in Section 3 and where

$$\begin{aligned} \phi_{pre} &= \text{closed}(D1) \wedge \text{closed}(D2) \wedge \text{closed}(D3) \wedge \neg\text{closed}(D4) \wedge \\ &\quad \text{in}(A,R1) \wedge \text{in}(B,R1) \wedge \text{in}(C,R1) \wedge \text{rob}(R1) \wedge \\ &\quad \neg \exists x \text{ holds}(x) \\ \phi_{post} &= \text{in}(A,R4) \wedge \text{in}(B,R4) \wedge \text{in}(C,R4) , \\ \text{INV} &= \mathbf{inv-in}(b,r) : (b \neq A \wedge b \neq B \wedge b \neq C) , \\ \phi_{safe} &= \neg \exists d_1 d_2 (d_1 \neq d_2 \wedge \neg\text{closed}(d_1) \wedge \neg\text{closed}(d_2)) . \end{aligned}$$

The initial situation as given by ϕ_{pre} is depicted in Figure 1. The robot has to carry blocks A, B, and C to room R4. In that process he is not allowed to change the position of any other block. As an additional safety condition we have that at most one door might be open in each situation.

At the top level, planning is done based on the assertions of the given abstract operations. In our example, Phase 1 comes up with the abstract plan “move*” introduced in Section 4 and we simply have to instantiate this general recursive solution. After the application of substitutions $s \leftarrow \{A,B,C\}$, $r_1 \leftarrow R1$, and $r_2 \leftarrow R4$ certain *proof obligations* arise. They guarantee that the concrete specification is met with respect to the abstract level, i.e. this particular instance of the abstract solution solves our planning problem modulo the concrete actions we will insert for the abstract ones. The proof obligations state: The preconditions ϕ_{pre} of the planning problem imply the preconditions of “move*”, i.e. “move*” is applicable in the current initial state:

$$\phi_{pre} \rightarrow \forall b (b \in \{A,B,C\} \rightarrow \text{in}(b,R1)) .$$

The postconditions of “move*” meet the current goals ϕ_{post} :

$$\forall b (b \in \{A,B,C\} \rightarrow \text{in}(b,R4)) \rightarrow \phi_{post} ,$$

and the proposed solution doesn’t violate the invariance conditions required, i.e. the invariance clauses of “move*” satisfy INV:

$$\mathbf{inv-in}(b,r) : b \notin \{A,B,C\} \rightarrow \mathbf{inv-in}(b,r) : (b \neq A \wedge b \neq B \wedge b \neq C) ,$$

In our example each of the formulae can be proved and we are ready to carry out the first refinement step

$$\varphi_0 \succ \neg\phi_{pre} \vee (\phi_{pre} \wedge \Box\phi_{safe} \wedge \text{move}^*({A,B,C}, R1, R2)) .$$

Please note that the safety conditions can not be guaranteed at this level and therefore still appear in the refined specification.

In the second phase we will perform a symbolic execution of the call $\text{move}^*(\dots)$. According to the “move*” assertions (cf. Section 4) we know that each way of unwinding the recursive procedure will stop with the desired result. That is all the unwinding can be done in a single step. However, since we have to make a choice about the order of movements, it is a better idea to unwind in a stepwise way. In cases where the order is relevant for the planning process on the lower level, backtracking to some intermediate situation will be possible this way. Following this strategy we get

$$\begin{aligned} &(\neg\phi_{pre} \vee (\phi_{pre} \wedge \Box\phi_{safe} \wedge \\ &\quad \text{STUT}_{move} ; \mathbf{delete-in}(A, R1) ; \text{STUT}_{move} ; \\ &\quad \mathbf{add-in}(A, R4) ; \text{STUT}_{move} ; \text{move}^*({B,C}, R1, R2))) \end{aligned}$$

as the next intermediate specification. Now we are in a situation where Phase 3, namely the actual planning process at the lower level, may start. Note that in most cases there will be no uniform way of filling out the missing steps. In our example A can be moved through door D4 while this is impossible for C. A possible strategy is to look for an action that deletes $\text{in}(A, R1)$. Applying the substitution $x \leftarrow A, r_1 \leftarrow R1, r_2 \leftarrow R4, d \leftarrow D4$ to W_1 we can identify “walk” as an appropriate action by matching the elementary operations given by W_1 against the intermediate specification above:

$$\begin{aligned} &\mathbf{delete-in}(A, R1) ; \underbrace{\mathbf{delete-rob}(R1) ; \mathbf{add-rob}(R4)} ; \mathbf{add-in}(A, R4) \\ &\mathbf{delete-in}(A, R1) ; \text{STUT}_{move} ; \mathbf{add-in}(A, R4) \end{aligned}$$

Since $\mathbf{delete-rob}(R1) ; \mathbf{add-rob}(R4)$ doesn't affect the in relation, i.e. implies STUT_{move} , we are allowed to replace the second occurrence of STUT_{move} by $\mathbf{delete-rob}(R1) ; \mathbf{add-rob}(R4)$ and then apply W_1 for further refinement, thus obtaining

$$\begin{aligned} &(\neg\phi_{pre} \vee (\phi_{pre} \wedge \Box\phi_{safe} \wedge (\text{STUT}_{move} ; \\ &\quad (\mathbf{rob}(R1) \wedge \text{connects}(D4, R1, R4) \wedge \neg\text{closed}(D4) \wedge \text{holds}(A) \wedge \\ &\quad (\text{small}(A) \vee \text{broad}(D4)) \wedge \text{walk}(R1, R4)) ; \text{move}^*({B,C}, R1, R2)))) \end{aligned}$$

as a new specification. Since it will turn out later in the refinement process that the third occurrence of STUT_{move} in the formula above can be replaced by **skip** we already omit it here in order to ease readability.

The planning process continues by analyzing the preconditions of “walk”. First of all their rigid part is considered. It turns out that $connects(D4, R1, R4) \wedge (small(A) \vee broad(D4))$ can be proved from the domain facts and with that there is no need to backtrack at this point. In a second step, the flexible part of the preconditions is matched against ϕ_{pre} . As ϕ_{pre} satisfies $\neg closed(D4)$ as well as $rob(R1)$ but no match is found for $holds(A)$ it is suggested to search for an action that has $holds(A)$ as an effect and leaves the relations $closed$, rob , and in untouched. Inspecting the effect descriptions leads to “pickup” and we carry out the next refinement step. Using the substitutions $b \leftarrow A$ and $r \leftarrow R1$ the preconditions of “pickup” coincide with ϕ_{pre} and we obtain as a new specification:

$$\begin{aligned} & (\neg\phi_{pre} \vee (\phi_{pre} \wedge \Box\phi_{safe} \wedge (pickup(A) ; \\ & \quad (rob(R1) \wedge connects(D4, R1, R4) \wedge \neg closed(D4) \wedge holds(A) \wedge \\ & \quad (small(A) \vee broad(D4)) \wedge walk(R1, R4)) ; move*({B,C}, R1, R2)))) \end{aligned}$$

This step is justified as we are able to prove $((in(A, R1) \wedge rob(R1) \wedge \neg \exists x holds(x) \wedge pickup(A)) \rightarrow STUT_{move})$ using P_{inv} .

In order to generate the first part of the refined plan we have to verify that $pickup(A)$ meets the safety conditions of the problem specification and finally we “compute” the facts that hold after the application of “pickup”. These facts comprise the effects of “pickup” and those parts of ϕ_{pre} which are invariant against “pickup”. We prove the formulae $(\phi_{pre} \wedge pickup(A)) \rightarrow \Box\phi_{safe}$ and $(\phi_{pre} \wedge pickup(A)) \rightarrow FIN \wedge \Box(\circ \mathbf{false} \rightarrow \phi_1)$, the latter one stating that $pickup(A)$ terminates and ϕ_1 holds afterwards, where ϕ_1 is $(in(A, R1) \wedge in(B, R1) \wedge in(C, R1) \wedge rob(R1) \wedge holds(A) \wedge closed(D1) \wedge closed(D2) \wedge closed(D3) \wedge \neg closed(D4))$. After another refinement step we therefore obtain

$$\begin{aligned} & (\neg\phi_{pre} \vee (pickup(A) ; \\ & \quad (\phi_1 \wedge \Box\phi_{safe} \wedge (walk(R1, R4) ; move*({B,C}, R1, R2))))). \end{aligned}$$

Proceeding with “walk” in the same way leads to the specification

$$\begin{aligned} & (\neg\phi_{pre} \vee (pickup(A) ; walk(R1, R4) ; \\ & \quad (\phi_2 \wedge \Box\phi_{safe} \wedge move*({B,C}, R1, R2)))) , \end{aligned}$$

where ϕ_2 is $(in(A, R4) \wedge in(B, R1) \wedge in(C, R1) \wedge rob(R4) \wedge holds(A) \wedge closed(D1) \wedge closed(D2) \wedge closed(D3) \wedge \neg closed(D4))$. From this specification refinement planning proceeds by unwinding $move^*$ again.

6 Related Work and Conclusion

Hierarchical problem solving and the refinement of abstract solutions are important methods for reducing costs in planning. They provide a significant restriction of the search space and help to generate plans in a goal-directed way [Ten91]. Therefore, many approaches to the creation of abstraction levels and to hierarchical planning have been developed, e.g. [Sac74, Sac77, Tat77, Wil88, BY94, Kno94, KU94, LG95]. They comprise two different principles. One is concerned with abstracting the state space. A hierarchy of abstraction levels is obtained by stepwise ignoring certain details of the planning domain [Sac74]. In recent approaches these abstractions are automatically generated [BY94, Kno94, LG95]. The second abstraction method combines several basic actions to more comprehensive abstract ones [Sac77, Tat77, Wil88], thereby obtaining some kind of macro operation. Hierarchical planning then proceeds by expanding these macros.

In our approach, the definition of abstract solutions is subject to the process of domain modeling. The reason is that in our environment abstract plans are often recursive. We have introduced a method for stepwise refining such recursive solutions. This allows in particular for the generation of completely different concrete solutions in each expansion of the recursive call, depending on the specific properties of the current objects involved.

As for the representation formalism our planning approach relies upon, there is some relation to the ADL framework of Pednault [Ped89]. Like ADL, our temporal planning logic uses elementary add- and delete-operations to describe basic actions and with that also follows the STRIPS idea of using add and delete lists for relations [FN71]. ADL provides a fixed form of action description schemata which immediately correspond to situation calculus axioms.

In our environment, the add- and delete-operations are basic elements of a programming language that is used to specify actions and plans. This programming language provides conditionals, nondeterminism, and also recursion. It is completely embedded into a logic as all programming language constructs are temporal logic formulae. This means in particular, that planning problems (specifications) as well as more or less abstract plans and actions can be flexibly treated on the same linguistic level. Among others, it is this feature which distinguishes our representation from formalisms like ADL, Dynamic Logic [Har79, Ros81], Situation Calculus [MH69], or Fluent Theory [MW87] and which motivated the introduction of yet another planning logic.

We have introduced an approach to hierarchical planning where abstract recursive plans are stepwise refined to concrete solutions. Each refinement step includes a formal correctness proof thus guaranteeing soundness of the entire procedure and with that correctness of the final solution. The planning logic we use provides a flexible means to reason about specifications and plans and finally offers the perspective of treating also nonterminating and concurrent systems.

References

- [BY94] F. Bacchus and Q. Yang. Downward Refinement and the Efficiency of Hierarchical Problem Solving. *Artificial Intelligence*, 71:43–100, 1994.
- [FN71] R.E. Fikes and N.J. Nilsson. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, 2(3/4):189–208, 1971.
- [GSS91] G. Ghassem-Sani and S. Steel. Recursive Plans. In J. Hertzberg, editor, *Proceedings of the European Workshop on Planning (EWSP-91)*, pages 53–63. LNAI 522, Springer, 1991.
- [Har79] D. Harel. *First Order Dynamic Logic*. LNCS 68, Springer, New York, 1979.
- [Kno94] C. A. Knoblock. Automatically Generating Abstractions for Planning. *Artificial Intelligence*, 68:243–302, 1994.
- [KU94] M. Kramer and C. Unger. A Generalizing Operator Abstraction. In C. Bäckström and E. Sandewall, editors, *Current Trends in AI Planning*, pages 185–198. IOS Press, Amsterdam, 1994.
- [Lam94] L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.
- [LG95] A. L. Lansky and L.C. Getoor. Scope and Abstraction: Two Criteria for Localized Planning. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 1612–1618. Morgan Kaufmann, San Mateo, 1995.
- [MH69] J. McCarthy and P. Hayes. Some Philosophical Problems from the Standpoint of Artificial Intelligence. In *Machine Intelligence 4*, pages 463–502. 1969.
- [MM94] A. Mokkedem and D. Méry. A Stuttering Closed Temporal Logic for Modular Reasoning about Concurrent Programs. In *Proc. ICTL-94*, pages 382–397, 1994.
- [MP91] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer, 1991.
- [MW87] Z. Manna and R. Waldinger. How to Clear a Block: Plan Formation in Situational Logic. *Journal of Automated Reasoning*, 3:343–377, 1987.
- [Ped89] E. Pednault. ADL: Exploring the Middle Ground Between STRIPS and the Situation Calculus. In *Proceedings of the 1st International Conference on Principles of Knowledge Representation and Reasoning (KR-89)*, pages 324–332, 1989.

- [Ros81] S. J. Rosenschein. Plan Synthesis: A Logic Perspective. In *Proc. IJCAI-81*, 1981.
- [RP86] R. Rosner and A. Pnueli. A Choppy Logic. In *Symposium on Logic in Computer Science*, pages 306–313. IEEE Computer Society Press, 1986.
- [Sac74] E. D. Sacerdoti. Planning in a Hierarchy of Abstraction Spaces. *Artificial Intelligence*, 5:115–135, 1974.
- [Sac77] E. D. Sacerdoti. *A Structure for Plans and Behavior*. North-Holland, 1977.
- [SB93] W. Stephan and S. Biundo. A New Logical Framework for Deductive Planning. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI-93)*, pages 32–38, 1993.
- [Tat77] A. Tate. Generating Project Networks. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence (IJCAI-77)*, pages 888–893. Morgan Kaufmann, San Mateo, 1977.
- [Ten91] J. D. Tenenbergh. Abstraction in Planning. In J. Allen, H. A. Kautz, R. N. Pelavin, and J. D. Tenenbergh, editors, *Reasoning About Plans*, pages 213–284. Morgan Kaufmann, Los Altos, 1991.
- [Wil88] D. E. Wilkins. *Practical Planning*. Morgan Kaufmann, 1988.