Deduction-based Refinement Planning

Werner Stephan and **Susanne Biundo**
German Research Center for Artificial Intelligence (DFKI)
Stuhlsatzenhausweg 3
D-66123 Saarbrücken, Germany
{stephan, biundo}@dfki.uni-sb.de

Abstract

We introduce a method of deduction-based refinement planning where prefabricated 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.

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 (Manna & Pnueli 1991; Biundo, Dengler, & Köhler 1992; Stephan & Biundo 1993). 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.

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 oppinion, which is shared by other authors as well (Manna & Waldinger 1987; Ghassem-Sani & Steel 1991), cannot be carried out in a fully automatic way. These abstract algorithmic solutions are in an efficient way refined to specific concrete ones.

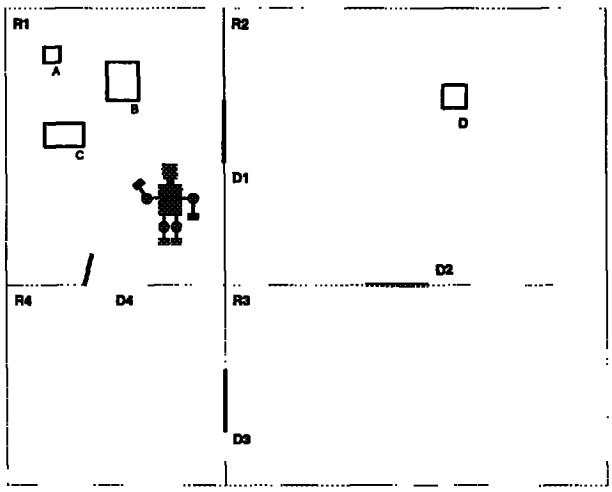

Figure 1: The 4–Room Planning Scenario

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 abstact 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.

The paper is organized as follows. In Section 2, we introduce our basic representation formalism, the temporal planning logic TPL. In Sections 3 and 4 it is shown how planning scenarios, planning problems, and abstract solutions are formulated in this framework. By means of a detailed example we demonstrate in Section 5 how refinement planning works in this context and we finally relate our work to existing approaches and conclude with some remarks in Section 6.

The Temporal Planning Logic TPL

We use an interval-based modal temporal logic—called TPL (*Temporal Planning 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 $X_z \cap (F_{f_{\epsilon,z}} \cup F_{r_{\epsilon,z}}) = \{\}$ 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:

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

$\bigcirc$ 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 $\Diamond\varphi :\leftrightarrow \neg\Box\neg\varphi$. In addition, the *strong next* operator is defined by $\odot\varphi :\leftrightarrow \neg\bigcirc\neg\varphi$. As can be seen from the semantics below, the *strong next* operator demands a next state to actually exist, while a formula $\bigcirc\varphi$ additionally holds in each one-state interval, where there is no next state. 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 $\beta : |\mathbf{X}| \to |\mathbf{D}|$. 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* (Rosner & Pnueli 1986). We use Σ^∞ to denote the set of all finite and infinite sequences $\bar{\sigma} =< \sigma_0, \sigma_1, \ldots >$ of states from Σ and $\Sigma^\bowtie$ for $\Sigma^\infty - \{< >\}$. By Σ^i we denote the set of sequences of length i. The concatenation of intervals is denoted by "$\cdot$". *Fusion*, denoted by "$\circ$", is the partial operation on intervals defined by:

$$\bar{\sigma} \circ \bar{\sigma}' = \begin{cases} \bar{\sigma} & \text{if } \bar{\sigma} \text{ is infinite} \\ < \sigma_0, \ldots \sigma_n, \ldots > & \text{if } \bar{\sigma} =< \sigma_0, \ldots \sigma_n >, \\ & \quad \bar{\sigma}' =< \sigma_n, \ldots > . \end{cases}$$

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 $[\![\varphi]\!]_{\mathcal{M},\beta} \subseteq \Sigma^\bowtie$. For atomic formulae ϕ containing only rigid symbols $[\![\phi]\!]_{\mathcal{M},\beta}$ is either $\Sigma^\bowtie$ or $\{\}$. The interpretation of the propositional connectives is also straightforward. The more interesting cases are as follows.

$$\begin{aligned}[\![r(\bar{t})]\!]_{\mathcal{M},\beta} &= \bigcup\{\{<\sigma>\} \cdot \Sigma^\infty \mid \\ &\qquad \sigma(r)([\![\bar{t}\,]\!]_{\mathcal{M},\beta})\}, \text{ for flexible } r \\ [\![\forall x.\varphi]\!]_{\mathcal{M},\beta} &= \bigcap\{[\![\varphi]\!]_{\mathcal{M},\beta(x/d)} \mid d \in D_z\}, \\ &\qquad \text{for } x \in X_z\end{aligned}$$

$$
\begin{aligned}
[\bigcirc\varphi]_{\mathcal{M},\beta} &= \Sigma^1 \cup (\Sigma^1 \cdot [\varphi]_{\mathcal{M},\beta}) \\
[\varphi_1 \,\mathcal{U}\, \varphi_2]_{\mathcal{M},\beta} &= \bigcup\{A_i \mid i \geq 0\}\ , \text{ where} \\
&\quad A_i = \Sigma^i \cdot [\varphi_2]_{\mathcal{M},\beta} \cap \\
&\quad \{\bigcap\{\Sigma^j \cdot [\varphi_1]_{\mathcal{M},\beta} \mid 0 \leq j < i\} \\
[\varphi_1 \,\mathcal{C}\, \varphi_2]_{\mathcal{M},\beta} &= [\varphi_1]_{\mathcal{M},\beta} \circ [\varphi_2]_{\mathcal{M},\beta} \\
[\gamma(\bar{t})]_{\mathcal{M},\beta} &= [\gamma]_{\mathcal{M},\beta}([\![\bar{t}\,]\!]_{\mathcal{M},\beta}) \\
[\textbf{add-}r(\bar{t})]_{\mathcal{M},\beta} &= \{<\sigma_0,\sigma_1> \mid \sigma_1 =_r \sigma_0\ , \\
&\quad \sigma_1(r) = \sigma_0(r) \cup ([\![\bar{t}\,]\!]_{\mathcal{M},\beta})\} \\
[\textbf{delete-}r(\bar{t})]_{\mathcal{M},\beta} &= \{<\sigma_0,\sigma_1> \mid \sigma_1 =_r \sigma_0\ , \\
&\quad \sigma_1(r) = \sigma_0(r) - ([\![\bar{t}\,]\!]_{\mathcal{M},\beta})\}
\end{aligned}
$$

For abstractions we have

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

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

In general, formulae are interpreted as sets of intervals. $\bar{\sigma} \in [\varphi]_{\mathcal{M},\beta}$ means that, given $\mathcal{M}$ and β, $\bar{\sigma}$ satisfies φ (or φ holds in $\bar{\sigma}$). A first-order formula ϕ holds in an interval iff ϕ is true under the interpretation of the flexible symbols given by the first state of $\bar{\sigma}$. $\bigcirc\varphi$ holds in $\bar{\sigma}$ iff either $\bar{\sigma}$ is an one-state interval or else φ holds in the interval which results from $\bar{\sigma}$ by removing the first state. $\bigcirc$ **false** thus denotes the set of one-state intervals. $\varphi_1 \,\mathcal{U}\, \varphi_2$ holds in $\bar{\sigma}$ iff there exists a suffix $\bar{\sigma}'$ of $\bar{\sigma}$ which satisfies φ_2 and φ_1 holds in all suffixes inbetween $\bar{\sigma}$ and $\bar{\sigma}'$. The *always* operator is used to express partial correctness assertions. The formula $\Box\ (\bigcirc$ **false** $\rightarrow\ \phi)$ holds in $\bar{\sigma}$ iff either $\bar{\sigma}$ is infinite or else the first-order formula ϕ is true in the last state of $\bar{\sigma}$.

$\mathcal{C}$ is the chop-operator. Basically, $\mathcal{C}$ allows to split a given interval into two subintervals. For example, $(\varphi \wedge (\Box\ (\bigcirc$ **false** $\rightarrow\ \phi)))\ ;\ \psi$ holds in $\bar{\sigma}$ iff either $\bar{\sigma}$ is infinite and satisfies φ or else $\bar{\sigma}$ can be split into $\bar{\sigma}'$ and $\bar{\sigma}''$, where $\bar{\sigma}'$ satisfies φ and the (first-order) formula ϕ holds in the last state of $\bar{\sigma}'$ (which is also the first state of $\bar{\sigma}''$) and where $\bar{\sigma}''$ satisfies ψ.

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

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

The class of formulae denoted by these abbreviations are called *plan formulae*. Please note that the **choose** construct essentially describes the existential quantification.

In addition, the following holds.

$$
\begin{aligned}
\textbf{add-}r(t_1,\ldots,t_n) &\rightarrow\ (\odot\bigcirc\textbf{ false} \wedge (\tilde{\phi} \rightarrow \bigcirc\phi)) \\
\textbf{delete-}r(t_1,\ldots,t_n) &\rightarrow\ (\odot\bigcirc\textbf{ false} \wedge (\hat{\phi} \rightarrow \bigcirc\phi)) \\
\nu a(\bar{x}).\pi(\bar{t}) &\leftrightarrow\ \pi[\bar{x}/\bar{t}][a/\nu a(\bar{x}).\pi]\ ,
\end{aligned}
$$

where $\tilde{\phi}$ is the formula resulting from the (first-order) formula ϕ by replacing each occurrence of the atomic subformula

$r(s_1,\ldots,s_n)$ by

$$\Big((t_1 \not\equiv s_1 \vee \ldots \vee t_n \not\equiv s_n) \rightarrow r(s_1,\ldots,s_n)\Big)\ .$$

$\hat{\phi}$ results from ϕ by replacing

$r(s_1,\ldots,s_n)$ by

$$\Big(r(s_1,\ldots,s_n) \wedge (t_1 \not\equiv s_1 \vee \ldots \vee t_n \not\equiv s_n)\Big)\ ,$$

respectively. $\tilde{\phi}$ and $\hat{\phi}$ are the weakest preconditions (in the sense of (Dijkstra 1976)) of ϕ with respect to **add-**$r(t_1,\ldots,t_n)$ and **delete-**$r(t_1,\ldots,t_n)$, respectively.

The equivalences therefore state that a first-order formula ϕ holds after the execution of an elementary add- (delete-) operation, if $\tilde{\phi}$ ($\hat{\phi}$) holds before.

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

$$
\begin{aligned}
\text{pickup}(b) \Leftarrow\ &\textbf{if } \exists\, r\ (in(b,r) \wedge rob(r)) \wedge \\
&\quad \neg\,\exists\, x\ holds(x) \\
&\textbf{then add-}holds(b) \textbf{ else skip fi}
\end{aligned}
$$

$$
\begin{aligned}
\text{putdown}(b) \Leftarrow\ &\textbf{if } \exists\, r\ (in(b,r) \wedge rob(r)) \wedge \\
&\quad holds(b) \\
&\textbf{then delete-}holds(b) \textbf{ else skip fi}
\end{aligned}
$$

$$
\begin{aligned}
\text{walk}(r_1,r_2) \Leftarrow\ &\textbf{if } rob(r_1) \wedge \exists\, d\ (connects(d,r_1,r_2) \\
&\quad \wedge \neg closed(d)) \\
&\textbf{then if } \exists\, x\ holds(x) \textbf{ then} \\
&\quad \textbf{choose } x : holds(x) \\
&\quad \textbf{begin if } small(x) \vee broad(d)
\end{aligned}
$$

then delete-*in*(x, r_1);
delete-*rob*(r_1);
add-*rob*(r_2);
add-*in*(x, r_2)
else skip fi end
else delete-*rob*(r_1);
add-*rob*(r_2) fi
else skip fi .

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. Please note that applying an action to inappropriate arguments leads to the trivial action **skip** rather than an "impossible" action, like **false**. The reason is that the impossible action may lead to *inconsistencies*, i.e. to formulae for which there is no satisfying interval. As a consequence, formulae stating certain properties of actions and plans, like partial correctness assertions, may trivially become true.

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]

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

$$W_2 : (rob(r_1) \wedge connects(d, r_1, r_2) \wedge \neg closed(d) \wedge \neg \exists x\, holds(x)) \rightarrow (walk(r_1, r_2) \leftrightarrow \textbf{delete-}rob(r_1); \textbf{add-}rob(r_2)) .$$

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. As for "walk" we obtain W_{eff1} and W_{eff2}, respectively. W_{eff1}, for example, reads:

$$W_{eff1} : (rob(r_1) \wedge connects(d, r_1, r_2) \wedge \neg closed(d) \wedge holds(x)) \rightarrow \Box\, (\bigcirc \textbf{false} \rightarrow (\neg in(x, r_1) \wedge \neg rob(r_1) \wedge rob(r_2) \wedge in(x, r_2)) .$$

For "pickup", for example, we obtain only one such axiom:

$$P_{eff} : (\exists r\, (in(b, r) \wedge rob(r)) \wedge \neg \exists x\, holds(x) \wedge pickup(b)) \rightarrow \Box\, (\bigcirc \textbf{false} \rightarrow holds(b)) .$$

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

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

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

$$P_{inv} : pickup(b) \rightarrow \Box\, (\textbf{inv-}in(b', r) : \textbf{true} \wedge \textbf{inv-}closed(d) : \textbf{true} \wedge \textbf{inv-}rob(r) : \textbf{true}) ,$$

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.

Note that we have specified the effects of basic actions in terms of *partial correctness assertions*, i.e. saying "whenever the action terminates, the effect holds afterwards". Note also that the $\Box$-operators in the effect and invariance formulae refer to just the intervals satisfying the respective action; they do not affect the entire interval satisfying the plan within which they occur. Let us give an example.

Suppose, we want to prove that the formula

$$\phi : \exists r\, (in(b, r) \wedge rob(r)) \wedge \neg \exists x\, holds(x)$$

holds after the execution of the plan $pickup(b)$; $putdown(b)$. This is expressed by

$$\phi \wedge (pickup(b) ; putdown(b)) \rightarrow \Box\, (\bigcirc \textbf{false} \rightarrow \phi) .$$

We begin with

$$\phi \wedge (pickup(b) ; putdown(b)) .$$

As ϕ is first-order it has to hold in the first state of the interval. Therefore, we obtain

$$(\phi \wedge pickup(b)) ; putdown(b) .$$

$pickup(b)$ terminates, i.e. we have $(\phi \wedge pickup(b)) \rightarrow \Diamond \bigcirc \textbf{false}$. From P_{eff} and P_{inv} it follows in addition that both the effect of $pickup(b)$ and the part of ϕ which is invariant against $pickup(b)$ hold in the last state of the interval satisfying $(\phi \wedge pickup(b))$. Since $pickup(b)$ and $putdown(b)$ are combined by ;, the last state of the "$pickup(b)$-interval" coincides with the first state of the "$putdown(b)$-interval". So, we obtain

$$pickup(b) ; (\exists r\, (in(b, r) \wedge rob(r)) \wedge \forall b' (b' \not\equiv b \rightarrow \neg holds(b')) \wedge holds(b) \wedge putdown(b)) .$$

Reasoning about $putdown(b)$ in a similar way, we finally succeed in proving the original assertion by deriving

$$\Box\, (\bigcirc \textbf{false} \rightarrow \phi) .$$

A possible format for *initial specifications* φ_0 is

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

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

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

$$\text{FIN} \quad :\leftrightarrow \ \Diamond\, \bigcirc\, \textbf{false}\ ,$$

$$\text{EF} \quad :\leftrightarrow \ \phi_{pre} \rightarrow \Box\, (\bigcirc\, \textbf{false} \rightarrow \phi_{post})\ ,$$

$$\text{SAFE} \quad :\leftrightarrow \ \phi_{pre} \rightarrow \Box\, (\phi_{safe_1} \wedge \ldots \wedge \phi_{safe_n})\ ,$$

$$\text{INV} \quad :\leftrightarrow \ \Box\, (\textbf{inv-}r_1(\bar{x}_1) : \phi_1(\bar{x}_1) \wedge \ldots \wedge \textbf{inv-}r_m(\bar{x}_m) : \phi_m(\bar{x}_m))\ .$$

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) .

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* $(s, r_1, r_2) \Leftarrow$ **if** $\forall\, b\ (b \in s \rightarrow in(b, r_1))$ **then**
choose $b : b \in s$
begin
move(b, r_1, r_2) ;
move*$(s - \{b\}, r_1, r_2)$
end else skip fi , where

move $(b, r_1, r_2) \Leftarrow$ **if** $in(b, r_1)$ **then**
delete-*in*(b, r_1) ;
add-*in*(b, r_2) **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 $\in$, $\emptyset$, $\{\ldots\}$, and $(-)$. The idea 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* (Lamport 1994; Mokkedem & Méry 1994). 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) in the body of move(b, r_1, r_2), for example, is replaced by[2]

$$\text{STUT}_{move}\ ;\ \textbf{delete-}in(b, r_1)\ ;\ \text{STUT}_{move}\ ,$$

where STUT_{move} is the formula

$$\Diamond\, \bigcirc\, \textbf{false} \wedge \Box\, \textbf{inv-}in(b', r) : (b \neq b' \vee r = r_1 \vee r = r_2)\ .$$

This allows for inserting certain steps between the essential add- and delete-operations. However, STUT_{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 STUT_{move} in both cases, we obtain the abstract plan move*(s, r_1, r_2). We are able to prove the assertions

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

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

$$\text{move*}(s, r_1, r_2) \rightarrow \forall\, b\ (b \in s \rightarrow in(b, r_1)) \rightarrow \Box\, (\bigcirc\, \textbf{false} \rightarrow \forall\, b\ (b \in s \rightarrow in(b, r_2)))\ ,$$

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.

Refinement Planning

In our environment, refinement planning consists in transforming an initial specification—via intermediate

[2] Note that b, r_1, and r_2 always correspond to the current arguments of the "move" call.

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

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

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

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

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 $s \leftarrow \{\mathrm{A,B,C}\}$, $r_1 \leftarrow \mathrm{R1}$, and $r_2 \leftarrow \mathrm{R4}$ 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:

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

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

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

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

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

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

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

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 move*(...). 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

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

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 $\mathit{in}(\mathrm{A,R1})$. Applying the substitution $x \leftarrow \mathrm{A}$, $r_1 \leftarrow \mathrm{R1}$, $r_2 \leftarrow \mathrm{R4}$, $d \leftarrow \mathrm{D4}$ 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:

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

Since **delete-***rob*(R1) ; **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 **delete-***rob*(R1) ; **add-***rob*(R4) and then apply W_1 for further refinement, thus obtaining

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

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 proces continues by analyzing the preconditions of "walk". First of all their rigid part is considered. It turns out that *connects*(D4, R1, R4) ∧ (*small*(A)∨*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}. Since ϕ_{pre} satisfies ¬*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:

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

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 $(\phi_{pre} \wedge pickup(A)) \rightarrow FIN \wedge \Box(\bigcirc \mathbf{false} \rightarrow \phi_1)$, the latter one stating that *pickup*(A) terminates and ϕ_1 holds afterwards, where ϕ_1 is (*in*(A,R1) ∧ *in*(B,R1) ∧ *in*(C,R1) ∧ *rob*(R1) ∧ *holds*(A) ∧ *closed*(D1) ∧ *closed*(D2) ∧ *closed*(D3) ∧ ¬*closed*(D4)). After another refinement step we therefore obtain

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

Proceeding with "walk" in the same way leads to the specification

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

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

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 (Tenenberg 1991). Therefore, many approaches to the creation of abstraction levels and to hierarchical planning have been developed, e.g. (Bacchus & Yang 1994; Knoblock 1994; Kramer & Unger 1994; Lansky & Getoor 1995; Sacerdoti 1974; 1977; Tate 1977; Wilkins 1988). 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 (Sacerdoti 1974). In recent approaches these abstractions are automatically generated (Bacchus & Yang 1994; Knoblock 1994; Lansky & Getoor 1995). The second abstraction method combines several basic actions to more comprehensive abstract ones (Sacerdoti 1977; Tate 1977; Wilkins 1988), 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 (Pednault 1989). 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 (Fikes & Nilsson 1971). 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 (Harel 1979; Rosenschein 1981), Situation Calculus (McCarthy & Hayes 1969), or Fluent Theory (Manna & Waldinger 1987) 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.

The procedure of refinement planning is currently implemented as part of a deductive planning system (Bauer *et al.* 1993) which generates plans from purely declarative specifications, based on a representation formalism similar to the one presented here.

Acknowledgement

We are indebted to the anonymous referees who carefully read the paper and whose comments led to a clear improvement of the presentation.
Our work has partly been supported by the Federal Ministry of Education, Science, Research and Technology under grant ITW 9404.

References

Bacchus, F., and Yang, Q. 1994. Downward Refinement and the Efficiency of Hierarchical Problem Solving. *Artificial Intelligence* 71:43–100.

Bauer, M.; Biundo, S.; Dengler, D.; Köhler, J.; and Paul, G. 1993. PHI – A Logic-Based Tool for Intelligent Help Systems. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI-93)*, 460–466. Morgan Kaufmann.

Biundo, S.; Dengler, D.; and Köhler, J. 1992. Deductive Planning and Plan Reuse in a Command Language Environment. In *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI-92)*, 628–632. Wiley.

Dijkstra, E. 1976. *A Discipline of Programming.* Prentice Hall.

Fikes, R., and Nilsson, N. 1971. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence* 2(3/4):189–208.

Ghassem-Sani, G., and Steel, S. 1991. Recursive Plans. In Hertzberg, J., ed., *Proceedings of the European Workshop on Planning (EWSP-91)*, 53–63. LNAI 522, Springer.

Harel, D. 1979. *First Order Dynamic Logic.* New York: LNCS 68, Springer.

Knoblock, C. A. 1994. Automatically Generating Abstractions for Planning. *Artificial Intelligence* 68:243–302.

Kramer, M., and Unger, C. 1994. A Generalizing Operator Abstraction. In Bäckström, C., and Sandewall, E., eds., *Current Trends in AI Planning*, 185–198. IOS Press.

Lamport, L. 1994. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems* 16(3):872–923.

Lansky, A. L., and Getoor, L. 1995. Scope and Abstraction: Two Criteria for Localized Planning. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI-95)*, 1612–1618. Morgan Kaufmann.

Manna, Z., and Pnueli, A. 1991. *The Temporal Logic of Reactive and Concurrent Systems.* Springer.

Manna, Z., and Waldinger, R. 1987. How to Clear a Block: Plan Formation in Situational Logic. *Journal of Automated Reasoning* 3:343–377.

McCarthy, J., and Hayes, P. 1969. Some Philosophical Problems from the Standpoint of Artificial Intelligence. In *Machine Intelligence 4.* 463–502.

Mokkedem, A., and Méry, D. 1994. A Stuttering Closed Temporal Logic for Modular Reasoning about Concurrent Programs. In *Proceedings of the 1st International Conference on Temporal Logic (ICTL-94)*, 382–397. Springer.

Pednault, E. 1989. 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)*, 324–332. Morgan Kaufmann.

Rosenschein, S. J. 1981. Plan Synthesis: A Logic Perspective. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence (IJCAI-81).* Morgan Kaufmann.

Rosner, R., and Pnueli, A. 1986. A Choppy Logic. In *Symposium on Logic in Computer Science*, 306–313. IEEE Computer Society Press.

Sacerdoti, E. D. 1974. Planning in a Hierarchy of Astraction Spaces. *Artificial Intelligence* 5:115–135.

Sacerdoti, E. D. 1977. *A Structure for Plans and Behavior.* North-Holland.

Stephan, W., and Biundo, S. 1993. A New Logical Framework for Deductive Planning. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI-93)*, 32–38. Morgan Kaufmann.

Tate, A. 1977. Generating Project Networks. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence (IJCAI-77)*, 888–893. Morgan Kaufmann.

Tenenberg, J. D. 1991. Abstraction in Planning. In Allen, J.; Kautz, H. A.; Pelavin, R. N.; and Tenenberg, J. D., eds., *Reasoning About Plans.* Morgan Kaufmann. 213–284.

Wilkins, D. E. 1988. *Practical Planning.* Morgan Kaufmann.