A Structured Approach for Synthesizing Planners from Specifications

Biplav Srivastava Subbarao Kambhampati
Amol D. Mali
Department of Computer Science and Engineering
Arizona State University, Tempe, AZ 85287-5406.
Email: {biplav,rao,amol.mali}@asu.edu WWW: http://rakaposhi.eas.asu.edu/yochan.html

Abstract

Plan synthesis approaches in AI fall into two categories: domain-independent and domain-dependent. The domain-independent approaches are applicable across a variety of domains, but may not be very efficient in any one given domain. The domain-dependent approaches can be very efficient for the domain for which they are designed, but would need to be written separately for each domain of interest. The tediousness and the error-proneness of manual coding have hither-to inhibited work on domain-dependent planners. In this paper, we describe a novel way of automating the development of domain dependent planners using knowledge-based software synthesis tools. Specifically, we describe an architecture called CLAY in which the Kestrel Interactive Development System (KIDS) is used in conjunction with a declarative theory of domain independent planning, and the declarative control knowledge specific to a given domain, to semi-automatically derive customized planning code. We discuss what it means to write declarative theory of planning and control knowledge for KIDS, and illustrate it by generating a range of domain-specific planners using state space and plan space refinements. We demonstrate that the synthesized planners can have superior performance compared to classical refinement planners using the same control knowledge.

1. Introduction

Planning is the problem of synthesizing a sequence of actions from a set of possible action templates such that when they are executed from the initial world state, all goal constraints are satisfied [4, 12]. Planning is known to be a combinatorial problem, and a variety of approaches for plan synthesis have been developed over the past twenty years. These approaches can be classified into two varieties – domain independent and domain dependent. Domain independent planners take a description of the actions, and the initial and goal state specifications, and produce a plan for the problem using the given actions. In contrast, domain dependent planners have either the actions or the initial or goal states hard-wired.

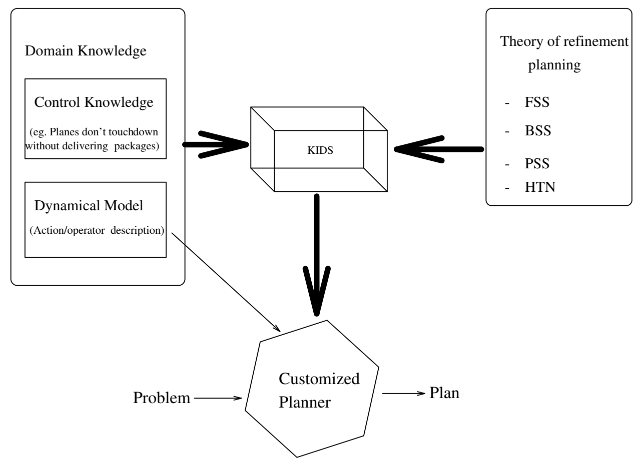

Figure 1. Architectural overview of planner synthesis with KIDS in the CLAY approach.

The domain-independent approaches are applicable across a variety of domains by simply changing the action template that is input to the algorithm, but may not be very efficient in any one given domain. The domain-dependent approaches need to be (re)designed for each domain separately, but can be very efficient in the domain for which they are designed as they can exploit the structure inherent in the given domain.

In this paper, we introduce CLAY as a novel approach for synthesizing efficient domain-dependent planners using automated software engineering tools leading to a fast turn-around time in implementing domain-dependent planners. In this work, as is shown in Figure 1, a declarative theory of plan synthesis (theory of planning) is combined with the control knowledge specific to a given domain, in a semi-automated software synthesis system called KIDS (Kestrel Interactive Development System) [17] to derive a customized planner for the domain. We will draw the declarative theory of plan synthesis from domain independent planning techniques. Domain specific information will be expressed in terms of the types of plans that are preferred in the given domain.

The advantage of the CLAY approach is that the theories of planning are encoded independent of domains, and the domain control knowledge can be encoded independent of the specific planning theory being used. The customization step compiles the domain control knowledge into the planning algorithm and ensures that the resulting planners are able to exploit the structure of the domain. Finally, the emphasis is shifted from "efficient plan generation" to "the generation of efficient planners".

Since our work straddles the areas of automated software engineering and AI planning, we have to be careful with our terminology for concepts that may be confused. The term *theory* refers to any useful body of knowledge. *Problem* refers to any assignment that is given to KIDS for solving and the successful outcome is called the *solution* to the problem. The input to KIDS is a problem theory consisting of the problem specification and a declarative description of useful concepts and rules to reason in the problem space. We give planning as a problem to KIDS and expect it to synthesize a planner which is the solution for us. The planner can take planning *tasks* as input and return *results*. For example, we can give the assignment of synthesizing a progression planner for blocks world domain to KIDS (the problem) and expect it to return such a planner (the solution). The synthesized planner can take a planning task such as Sussman Anomaly[1] as input (a task) and return the plan for it (the result).

1.1. Background

Part of the reason for our interest in KIDS system stems from the fact that it has been used in the past to derive efficient scheduling software [16]. Using KIDS to derive planning software in CLAY involves figuring out (a) how declarative theories for different types of classical planning are specified and (b) what algorithmic design templates are best suited to planner synthesis. We answer these questions with the help of a unifying framework for plan synthesis algorithms (called Universal Classical Planning or UCP) that we have developed in our recent work [9, 8]. In this framework, plan synthesis is modeled as a process of searching in a space of sets of action sequences. These sets are represented compactly as collections of constraints called "partial plans." The search process first attempts to extract a result (an action sequence capable of solving the task) from the partial plan, and when that fails, "refines" (or splits) the partial plan into a set of new partial plans (each corresponding to sets of action sequences that are subsets of the original partial plan's action sequence set), and considers the new plans in turn. As mentioned in Kambhampati and Srivastava [10], existing domain independent plan-synthesis algorithms correspond to four different ways of refining partial plans. These are known, respectively, as Forward State Space Refinement (FSS), Backward State Space Refinement (BSS) e.g. TOPI, Plan Space Refinement (PSS) e.g. SNLP and Task Reduction Refinement (HTN) e.g. NONLIN [7]. Given this background, the declarative theory of plan generation in CLAY corresponds to theories of these refinements. The algorithm tactic underlying plan generation corresponds to "refinement search." KIDS system supports an algorithm tactic called "global search" [17] which can be seen as a generalization of this refinement search.

As we give planning as a problem to KIDS, we have to develop a theory of planning (*planning theory*). Our view of planning envisages a common plan representation over which multiple refinements can describe how the partial plans are refined. Consequently, our planning theory should consist of a specification of planning and one or more refinement theories. Moreover, we are interested in domain dependent planners and so, we have to provide the necessary domain theory to KIDS.

Given these inputs, KIDS semi-automatically synthesizes a solution (domain-dependent refinement planner) using generic algorithm design tactics (such as branch and bound algorithms, global search algorithms). The resulting planner, like conventional planners, can handle any planning task from the domain. If no domain theory was given, the resulting planner can handle any planning task from any domain.

1.2. Outcome of our work

To understand the efficacy of plan synthesis in CLAY, we initially concentrated on the synthesis of planners using state-space refinement theories (FSS and BSS). Work on the synthesis of PSS planner is still in progress and we discuss our approach towards the end of this paper. Empirical evaluation shows that synthesized planners can be very efficient [18]. For example, in the blocks world domain where the goal was stack inversion, a KIDS synthesized planner could solve 14 blocks task in under a minute. This is unheard of in traditional planners. In the logistics domain, a task with 12 packages, 4 planes and 8 places was solved in under a minute. Similarly, in the Tyre domain [15], the fixit task was solved in under a minute. To put the performance results in perspective, we compared KIDS' synthesized planners and the instantiations of UCP (which emulate a spectrum of classical planners, including the popular SNLP planner [12], by selecting the appropriate refinement) across many blocks world tasks [18]. In our experiments with state-space planners for the blocks world domain, the best of the KIDS' synthesized planners outperformed the best of the UCP instantiations when given the same domain-specific informa-

[1] Sussman Anomaly refers to a task in blocks world domain with three blocks A, B and C. In the initial state, Block C is on top of Block A and Block B is on Table. The goal state has Block A on top of Block B, Block B on top of Block C and Block C on top of Table.

tion. We hypothesize that this is because KIDS can profitably fold-in the domain-specific control knowledge (i.e. the domain theory) into the planning code.

The paper is organized as follows: after a brief review of software synthesis on KIDS in Section 2, we walk-through the CLAY framework in Section 3 in the context of state space planners. Section 4 summarizes some of our empirical results (See detailed report in [18]). Section 5 presents our on-going effort to synthesize PSS planners within the CLAY framework. We cover related work in Section 6 and concluding comments in Section 7.

2. Background on KIDS

KIDS is a program-transformation framework for the development of programs from formal specifications of a problem. KIDS runs on Sun workstations and it is built over REFINE. The REFINE language supports first-order logic, set-theory, pattern matching and transformation rules. Refine has its own compiler that generates Common Lisp or C code. In the following, we describe the general steps involved in synthesizing software on KIDS. The process is illustrated in more detail in Section 3 in the context of synthesis of customized planner code.

1. *Develop a problem theory* to state and reason about the problem. The user defines appropriate functions and types that describe the problem and also gives laws that allow high-level reasoning about the defined functions.

2. *Apply a design tactic* to select an algorithmic framework that should be used to implement the problem specification.

3. *Apply optimizations* to make the generated algorithm efficient. The algorithm is optimized through simplification, partial evaluation, finite-differencing, etc.

4. *Compile* the algorithm to produce a software in the base language.

3. Developing a planner from declarative specification

Figure 1 summarizes how KIDS is used to synthesize a domain-specific refinement planner. Refinement planning was explained briefly in Section 1.1.

The domain knowledge consists of a dynamical model and control knowledge. The dynamical model is in the form of actions or operators that define legal transformation from one plan-state to another. For state space planners, the plan state is the world state. Control knowledge is a set of domain-specific criteria that helps the planner decide if a plan P_1 is better than P_2 and is intended to make search more efficient. An example of control knowledge is that in a logistics domain where some packages have to be moved to their destinations using airplanes, planes should not touch-down at a location if they have no package to deliver.

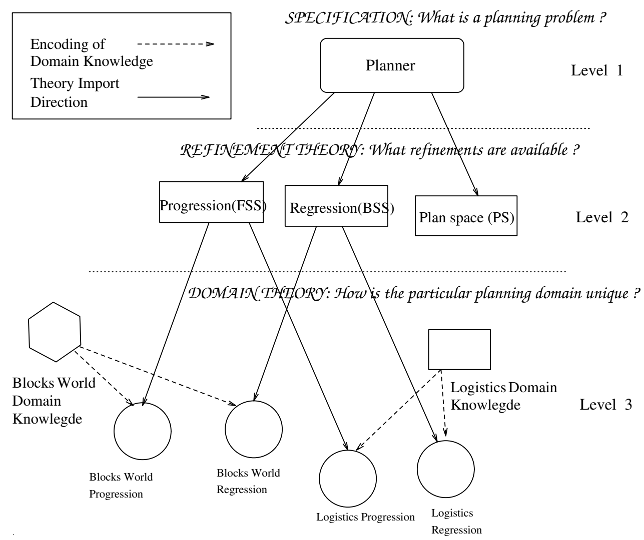

Figure 2. The CLAY architecture for writing planning theory.

The CLAY Architecture: Refinement planning and domain control knowledge are brought together in the CLAY architecture for writing declarative domain-specific planning theory as summarized in Figure 2. To specify a planning task, a plan representation is selected and all behaviors that a result plan of a planning task must show are enumerated in the planner specification. The planner *specification* is dependent on the plan representation but is independent of the refinement needed for search. A *refinement* for planning uses the planner specification and defines how children nodes are generated, what will be the goal test and any refinement specific search pruning test. The refinement and the specification together form the *planning theory*. To obtain a domain-dependent planner, all one needs to do is import any planning theory and provide any relevant planning domain-specific control knowledge (*domain theory*). An interesting special case is when one specifies a *generic* domain knowledge to the effect that all the plans are good in the "domain". In such a case, based on the refinement used, one gets a FSS, BSS, PSS or hybrid (if multiple refinements are used), general-purpose planner.

Each level in the directed tree in Figure 2 represents an abstraction of the planning task. At the root of the tree (Level 1), only a description of a planning task is required without specifying what refinements strategies should be used. At Level 2, the refinements are specified but no assumption about the domain is made. Next, characteristics of the domain are provided at Level 3. A progression (FSS) blocks world planner is different from a progression (FSS) logistics domain planner only in terms of the domain knowl-

STRIPS representation

- Action: move(?x, ?y, ?z)
- Pre: (?x != ?y) ∧(?y != ?z) ∧(?z != ?x) ∧clear(?x) ∧clear(?z) ∧¬ clear(?y) ∧on(?x, ?y)
- Post: on(?x,?z) ∧clear(?y) ∧¬ clear(?z) ∧¬ on(?x, ?y)

Put ?x = A, ?y = B, ?z = C

State-variable representation
State = ≺ pos-A, clr-A, pos-B, clr-B, pos-C, clr-C ≻

- Pre: ≺ B, True, 0, False, 0, True ≻
- Post: ≺ C, True, 0, True, 0, False ≻

where 0 represents don't care.

Figure 3. Different representations of "Move A from B to C"

edge. On the other hand, a progression (FSS) blocks world is different from a regression (BSS) blocks-world planner only in terms of the refinement used.

3.1. Representing Domain Operators

We now discuss how a *world state* is represented and how the domain operators define state transformation from one state to another. The two approaches which are popular in the planning community are the STRIPS representation [4] and the multi-valued state-variable representation [1].

In the STRIPS representation, the plan state is represented as a clause of predicates expressing some information about the world. Actions have pre-conditions which specify when the action can be executed and post-conditions which express change to the plan state when the action is executed. In multi-valued state-variable representation, the attributes of the world objects are expressed by variables that can take a range of values (variables with binary values is a special case). Both of these representations are completely equivalent in their expressiveness. To us, the state-variable representation was more natural for expressing the world facts using the in-built data types of KIDS. Figure 3 shows the action of moving block A from block B to the top of block C in STRIPS and state-variable representation. A blocks world domain is an environment in which some blocks are placed on a table or on top of other blocks and the tasks involve stacking them in some desired configuration. For the purpose of exposition, we are showing values of state variables corresponding to block positions (e.g. pos-A) by symbols 'B', 'C', etc. and clear conditions (e.g. clr-A) by True or False. In practice, we map all the valid values of state variables to integers.

3.2. Specification of a planner

A specification of the problem ([17]) is represented by a quadruple $F = \langle D, R, I, O \rangle$ where D is the input type satisfying the input condition, $I : D \rightarrow boolean$. The output type is R and the output condition, $O : D \times R \rightarrow boolean$ defines a feasible solution. If *O(x,z)* holds, then z is a feasible solution with respect to input x. The specification of a program follows the template:

function *F* (x :*D*) : *set*(*R*)
where *I*(x)
returns { *O* (x,z) }
= *Body(x)*

A specification for program F is consistent if for all possible inputs satisfying the input condition, the body produces the feasible solution, i.e., $\forall(x{:}\ D)(I(x) \Longrightarrow O(x, z))$.

Within this view, a planner takes as inputs an initial state, a goal state and an operator list. The operators are assumed to define state transitions from valid states to valid states. A specification for the planning task is: given the initial state, the goal state and the operator list, return a sequence of operators (plan) such that:

- TERMINATION-TEST: The goal state can be achieved if the plan is executed. We are only considering planning problems in which the goal is to make all state-variables achieve specified values, i.e. goals of achievement.
- DOMAIN-INDEPENDENT-PRUNING-TEST: The plan passes the domain-independent pruning tests. Each planning refinement can specify conditions under which a partial-plan cannot have a minimal solution and such a plan can be pruned away. For example, in FSS refinement, if the plan-state after executing operator O_2 is a subset of the state following an earlier operator O_1, this partial-plan can be pruned. This is the idea of *state-looping* based pruning in classical state-space planning.
- DOMAIN-DEPENDENT-PRUNING-TEST: The plan passes the additional domain-specific pruning tests.

The above specification of planning is declarative in that it states what constraints must be satisfied in the resulting plan produced by a planner when given a planning task. It does not suggest any algorithm about how the results could be obtained. Algorithmic decisions will be made in the program development phase of KIDS.

An example of top-level specification of planning (in REFINE) is shown in Figure 4. In this specification, I (the input condition) is true, D (the input data type) is INIT, GOAL and OPERS, R (the output data type) is PLAN and O (the output condition) consists of GOODNESS-TEST, GOAL-TEST, NO-MOVES-BACK and the check on range of PLAN. GOAL-TEST and NO-MOVES-BACK are domain independent pruning tests whereas GOODNESS-TEST is a domain dependent pruning test.

```
function PLANNER
  (INIT:  seq(integer), GOAL: seq(integer),
   OPERS: seq(tuple(seq(integer), seq(integer))))
  returns
    (PLAN: seq(integer)
       | range(PLAN) subset {1 .. size(OPERS)}
         & GOODNESS-TEST(VISITED-STATES
                           (PLAN, INIT, GOAL, OPERS),
                         INIT, GOAL)
         & NO-MOVES-BACK(VISITED-STATES
                           (PLAN, INIT, GOAL, OPERS),
                         INIT, GOAL)
         & GOAL-TEST(VISITED-STATES
                       (PLAN, INIT, GOAL, OPERS),
                     INIT, GOAL))
```

Figure 4. A declarative specification for planning

In words, the specification says that a partial plan is a sequence of integer indices (of operators) and so the indices must not be more than the size of operator list. A valid plan is one whose corresponding state sequence (produced by VISITED-STATES) satisfies the GOAL-TEST, NO-MOVES-BACK and GOODNESS-TEST. In the context of FSS refinement, VISITED-STATES returns the states obtained by the successive application of the operators in the partial plan to the initial state and the resulting states thereafter. GOAL-TEST signals that the goal has been achieved; for FSS refinement it involves checking that the last state in the state-sequence is the goal state. The NO-MOVES-BACK function tests forward state-space looping.

The GOODNESS-TEST function checks for possible redundancy in the state sequence corresponding to the current partial plan based on domain characteristics. Let us explain it in the context of blocks world domain. We can specify any reasonable checks for the blocks world as long as it does not make the planner lose a result. Below, we present two GOODNESS-TESTs:

- (H1: *Limit useless moves*) If a block moves between states i and (i+1), it must not change position between (i+1) and (i+2) state. The motivation behind this check is to prevent blocks from randomly moving around.

- (H2: *Move via table*) A block can only move from its initial state to the table and from table only to its goal position. This check guides the planner to first put all the blocks on the table that are not already on the table, and then arrange blocks according to the goal configuration.

The problem specification is not complete without distributive laws. KIDS has a directed-inference engine called RAINBOW which uses the specification and laws specified by the user to simplify and reformulate the expressions. We specify such laws for all the operations involved in the planning specification.

```
 0. Focus Initialize PLANNER
 1. Tactic Global Search on PLANNER
 2. Simplify, context-independent-fast:
       if ## then ## else some (PLAN-2: ##...
 3. Simplify, context-dependent, forward-0,
       backward-4: ##(...) & ##(....
 4. Simplify, context-dependent, forward-0,
        backward-4: if ## else und...
 5. FD (general-purpose) VS =
       VISITED-STATES(V, INIT, GOAL, OPERS)
 6. FD (general-purpose) L-VS = last(VS)
 7. FD (general-purpose) NUM-OPS = size(OPERS)
 8. Abstract NEXT-STATE(L-VS, I, OPERS) into NS
       in ex (I: integer) (## in ## &...
 9. Simplify, context-independent-fast:
       if ## & ## then PLANNER-AUX(##, ##,...
10. Refine compile into Lisp: PLANNER-AUX, PLANNER
```

Figure 5. Derivation steps to generate a progression planner for blocks world domain.

3.3. From Specification to Code with KIDS

As discussed in Section 2, we need to select an algorithm design tactic to implement the specification. One of the design tactics provided by KIDS is *global search*. Global search (GS) is a general case of refinement search that we used to formalize refinement planning in UCP framework. GS algorithms work as follows: starting from an initial set that contains all solutions to the given problem instance, the algorithm repeatedly extracts solutions, splits sets and eliminates sets via filters until no sets remain to be split. The process can be described as a tree search in which a node represents a set of candidates and an arc represents the split relationship between a set and its subset. For complete details, readers are referred to [17].

Based on the problem specification, KIDS lists precanned global search theories to which it can automatically map the current problem instance using deductive inference. Since our data type was a sequence, we used the specialization of global search for sequences over a finite domain.

The first code produced by KIDS after incorporating a design tactic (in our case global search) is well-structured but very inefficient. There are several opportunities for optimization and KIDS provides tools for program optimization. These tools make use of the distributive and monotonic laws to simplify and optimize the code. Figure 5 shows a summary of the sequence of derivation steps carried out to obtain a blocks world domain-specific forward-state space planner.

In step 0, the top-level planner specification is selected and algorithmic design is performed in step 1. Step 2 involves a context independent simplification, and context dependent simplifications are done in steps 3 and 4. Steps 5 through 8 cover finite differencing. Finally, an efficient planner code is compiled in step 10 (shown in Figure 6).

```
function PLANNER-AUX
 (INIT: seq(integer), GOAL: seq(integer),
  OPERS: seq(tuple(seq(integer), seq(integer))),
  V: seq(integer), VS: seq(seq(integer)),
  NUM-OPS: integer, L-VS: seq(integer)
 | SEQEQUAL(L-VS, last(VS))
   & SEQEQUAL(VS,
       VISITED-STATES(V, INIT, GOAL, OPERS))
   & NO-MOVES-BACK
       (VISITED-STATES(V, INIT, GOAL, OPERS),
       INIT, GOAL)
   & GOODNESS-TEST
       (VISITED-STATES(V, INIT, GOAL, OPERS),
       INIT, GOAL)
   & range(V) subset {1 .. size(OPERS)}
   & NUM-OPS = size(OPERS))
 : seq(integer)
 = if GOAL-TEST(VS, INIT, GOAL) then V
    else some (PLAN-2: seq(integer))
     ex (NS: seq(integer), I: integer)
       (NS = NEXT-STATE(L-VS, I, OPERS)
       & CROSS-NO-MOVES-BACK(VS, [NS], INIT, GOAL)
       & CROSS-GOODNESS-TEST(VS, [NS], INIT, GOAL)
       & DEFINED?(PLAN-2)
       & PLAN-2
          = PLANNER-AUX
             (INIT, GOAL, OPERS, append(V, I),
              append(VS, NS), NUM-OPS, NS)
           & I in {1 .. NUM-OPS})

function PLANNER
 (INIT: seq(integer), GOAL: seq(integer),
  OPERS: seq(tuple(seq(integer), seq(integer))))
 returns
  (PLAN: seq(integer)
   | range(PLAN) subset {1 .. size(OPERS)}
    & GOODNESS-TEST
       (VISITED-STATES(PLAN, INIT, GOAL, OPERS),
       INIT, GOAL)
    & NO-MOVES-BACK
       (VISITED-STATES(PLAN, INIT, GOAL, OPERS),
       INIT, GOAL)
    & GOAL-TEST
       (VISITED-STATES(PLAN, INIT, GOAL, OPERS),
       INIT, GOAL))
 = PLANNER-AUX
    (INIT, GOAL, OPERS, [], [INIT],
     size(OPERS), INIT)
```

Figure 6. Final progression blocks world planner code synthesized by KIDS

4. Empirical evaluation of synthesized state space planners

In this section, we demonstrate that the CLAY approach for domain-specific planner synthesis is flexible as well as potentially superior than traditional planners. We conducted empirical study to confirm our hypothesis that since the domain control knowledge is folded into the planning process in CLAY while it is explicitly invoked at each iteration in classical planners, synthesized planners should outperform classical planners given the same control knowledge.

Name	Domain	Refinmt.	Domain Dependent Pruning Test
BW-P-H1	Blocks World	FSS	Limit useless moves (H1)
BW-P-H2	Blocks World	FSS	Move via table (H2)
BW-R-H1	Blocks World	BSS	Limit useless moves (H1)
BW-R-H2	Blocks World	BSS	Move via table (H2)
LOG-P-L	Logistics	FSS	Limit inefficiency
LOG-R-L	Logistics	BSS	Limit inefficiency
TYR-P-M	Tyre World	FSS	Multiple control rules
INDEP-P	******	FSS	-none-
INDEP-R	******	BSS	-none-

Table 1. Table showing the variety of planners synthesized on KIDS.

4.1. Domains and problems

Table 1 lists several domain-dependent state-space planners that we have synthesized until now. The planners are characterized by the domain for which they are developed (BW for blocks world, LOG for logistics, and TYR for Tyre World – all of which are benchmark domains in AI planning); the type of (state-space) refinement used (P for progression and R for regression), and the type of domain specific control knowledge used (H1, H2, etc.).

A blocks world domain is an environment in which some blocks are placed on a table and the tasks involve stacking them in some desired configuration. The logistics domain consists of a certain number of planes and packages at different places. The goal is to find a sequence of actions such that all planes and packages are at the goal positions. In the Tyre world [15], there is a car with spare tyre and tools in the boot. Given some tyre trouble, it may be inflated or replaced with the help of the tools. We used the same domain description as used in Graphplan [3].

4.2. Performance of Synthesized planners

The synthesized state space planners were able to solve large blocks world, logistics and Tyre world problems very efficiently. As can be seen from (Figure 7,left), the domain-specific blocks world with H1 (Limit useless move) helped the progression planner (BW-P-H1) solve the stack inversion task for 22 blocks in under 30 minutes (14 blocks task in under a minute). Similarly, in the logistics domain, the progression planner with Limit Inefficiency (LOG-P-L) could solve 6 plane task in around 30 minutes (4 plane task in under a minute) (Figure 7, right).

There are 25 operators, 27 state variables and 6 control rules in our manually encoded Tyre world description. The fixit task was solved in under a minute and a 31 step plan was returned. All these results are significantly better than the performance of traditional domain-independent classical planners on these benchmark domains.

Since our synthesized planners used domain specific control knowledge that is not normally used by domain-independent planners, our next step involved comparing

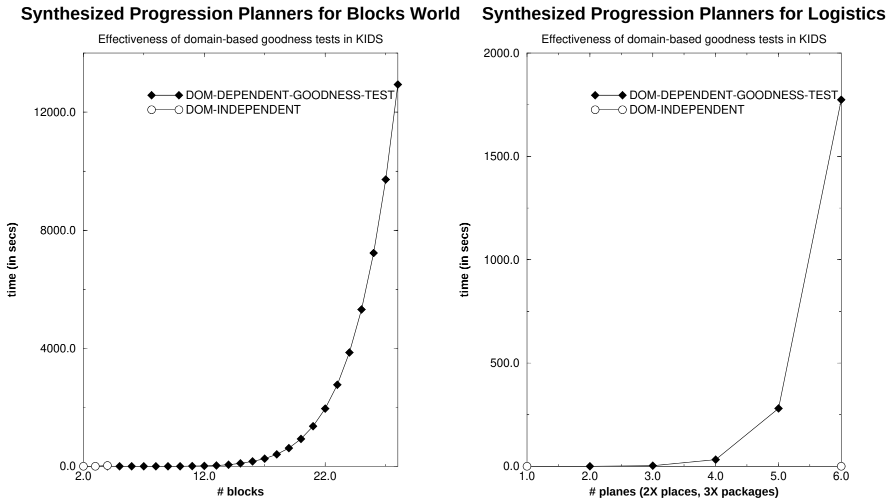

Figure 7. Effect of domain-dependent goodness tests on performance.

synthesized planners to domain-independent planners using the same control knowledge. Our methodology is to select a planning task suite in the blocks world domain and first experiment with multiple synthesized planners, each differing in the employed refinement or the domain control knowledge or both. Next, we run the task suite with a set of traditional planners obtained by instantiating UCP and selecting the best traditional planner. Finally, we compare the best of the synthesized planners for different tasks with the selected traditional planner.

We selected two task suites: random block world tasks and stack building task (See [18] for detail). Each task class is defined in terms of the number of blocks and an average of 10 runs is shown in each plot. The total time allowed for a class of tasks was 1000 seconds after which the planner was deemed to have failed on that task class. All planners were run on the same tasks from the task suite.

We experimented with different instantiations of UCP and determined that UCP-FSS is the best UCP strategy for blocks world domain. Similarly, we experimented with different KIDS' synthesized planners and found the best synthesized planners to be BW-P-H2 for the random blocks world task set and BW-R-H2 for the stack building task set. We next tested how UCP-FSS performs against the best of KIDS' synthesized planners. Comparison is done when all planners are either given the same heuristic information (H2) or no domain dependent guidance. Figure 8 plots the results.

In the left plot, UCP-FSS-DOM-INDEP does better than INDEP-P i.e. without any heuristic information. When H2 heuristic is given to both the planners, BW-P-H2 is a winner. In the right plot, BW-R-H2 outperforms UCP-FSS with H2. So, we see that *given the same heuristic information, the best of KIDS' synthesized planners can outperform the best instantiation of UCP for the blocks world.*

It is interesting to note that while all synthesized planners improve drastically with domain specific knowledge, domain independent planners don't always improve the same way. In fact, in Figure 8 all synthesized planners improve with H2, while the performance of UCP-FSS (the plot on the right) degrades. We speculate that this is because UCP-FSS explicitly calls a function to do domain-specific reasoning in each recursive invocation while the synthesized planners have domain control knowledge folded into the planner code.

5. Progress on Synthesizing Plan Space planners within CLAY

Until now, we have concentrated on synthesis of state-space planners. In this section we briefly describe our progress to-date on synthesizing plan-space planners [7] Plan space planners search in the space of partially ordered plans, introducing actions without restricting their position. Such an approach is considered to be more efficient than state space planning [2, 7]. Consequently, we have been working on synthesizing domain-dependent plan space planners.

In plan space refinement (PSR), a partial-plan is defined as a collection of steps, precedence constraints ($s_i \prec s_j$) between steps and causal links that indicate that pre-condition C of a step s_j is supported by step s_i ($\langle s_i, C, s_j \rangle$). The first difficulty in synthesizing plan-space planners is that the underlying datastructures are more complex than those in state space planners. For example, while the state space plans can be described as ranging over sequences of integers, plan space plans cannot be described in that way. In our current approach, we generate all possible causal links and precedence orderings *a priori*, and store them in a sequence. This sequence serves as the range of the solution for plan space

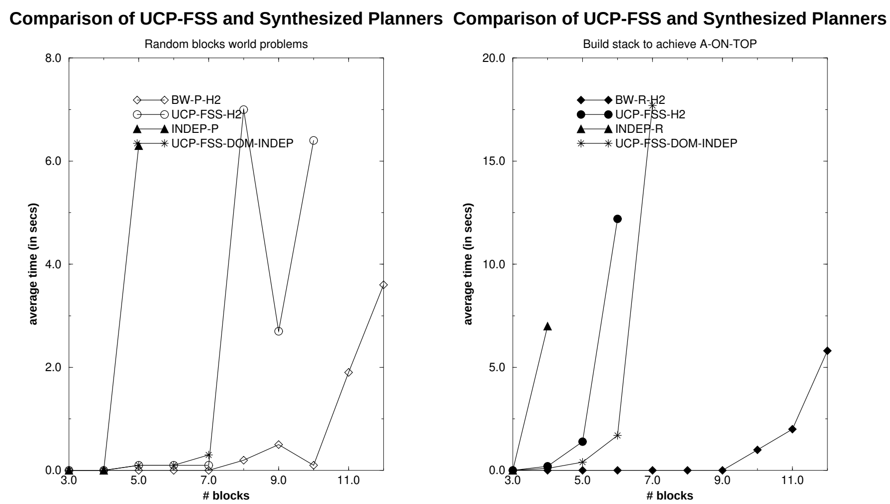

Figure 8. Comparing the best UCP strategy for blocks world – UCP-FSS – against the best of KIDS' synthesized planners.

planning. A sequence of some elements from this range meeting the specifications corresponds to the desired plan space plan.

The plan space plan is represented by the data type *seq(tuple(integer, seq(integer), integer))*. An example of an element of such a sequence is the tuple $(3(4000)5)$, meaning that there is a causal link between steps 3 and 5, where a precondition (represented by the state vector (4 0 0 0)) of step 5 is contributed by the effect of step 3. A precedence constraint demanding step 3 to come before step 5 is encoded as $(3(0000)5)$. The sequence of all zeros between step indices distinguishes an ordering from a causal link. This approach allows us to use the same data type $tuple(integer, seq(integer), integer)$ to represent both causal links and precedence orderings.

The second difficulty in synthesizing plan-space planners is that control knowledge is harder to specify for them. The type of pruning knowledge that we discussed in Section 4.1 is less effective for plan-space planners which do not maintain the current state of the world explicitly. At the time of this writing, we have synthesized a prototype planner for a simple logistics domain. To improve its efficiency, we are currently working on providing more effective search control during the synthesis process.

6. Related Work

The research reported here straddles the two fields of automated software synthesis and AI planning. Although there has not been much work on automated planner synthesis, Gomes [6] has synthesized a state-space problem solver for the "missionaries and cannibals" problem on KIDS, and has shown that it outperforms general purpose problem solvers in that domain. Our framework can be seen as a generalization of the work done by Gomes. In particular, we separate planning theories from the dynamics and control knowledge, which in principle supports generation of planning code based on a variety of refinements. Methodologically, our work adds to Gomes' results in that we have shown that given the same control knowledge, planners generated by KIDS can outperform traditional planners. This makes for a fairer comparison between synthesized and general-purpose planners.

There are many research projects in constraint satisfaction systems, for example, COASTOOL [19] and ALICE system [11] that take declarative description of CSPs and compile specialized algorithms for solving them. MULTI-TAC [13] does the same thing but uses distribution-oriented information and machine learning techniques to assist in the customization. We use declarative knowledge about refinement and planning domain to customize the planner.

7. Discussion and Conclusion

In this paper, we described the CLAY architecture for synthesizing efficient domain-dependent classical planners from declarative theory of planning and domain theory using a software synthesis system (KIDS). We have synthesized state space and shown that the synthesized planners can be superior to traditional planners when given the same domain control knowledge. We have also described the status of our efforts to synthesize plan-space planners. Even though newer breed of classical planners (e.g. Graphplan [3]) can solve the same problems in better or comparable time, in theory, even these planners can be synthesized within the CLAY architecture. Thus, the results give empirical evidence that the CLAY approach is an interesting research direction for obtaining efficient planners.

Our synthesis approach provides an interesting contrast to main-stream AI planning work in several ways. Most of AI planning work attempts to improve the efficiency of planning by concentrating on the way plans are generated. Our work sets the next stage in that we concentrate on how "efficient planners" may be synthesized. Generating planners by deductive methods requires reasoning about the logical aspects of planning and this is the functionality that the CLAY architecture promises.

The planning theory is specified declaratively rather than in the form of an implemented program. This supports changes and extensions to planning theory. Since the algorithm is synthesized from specification, the user is freed from low-level coding, and can concentrate on "declarative specification" of control knowledge (e.g. pruning tests) and the way the operations mentioned in the specification combine (e.g. distributive laws).

Additionally, in a typical classical planner like SNLP [12], UCPOP [14] and PRODIGY [5], domain control rules may be used to guide the search if the implementation has *programming hooks* at different choice points to make use of them. But such hooks need to be explicitly encoded in the planning algorithm and no context dependent analysis is done to optimize the planner (at code level) based on the available control knowledge. In contrast, we have encoded the information declaratively and the planning algorithm can be optimized based on *all* the knowledge that is available, including the control knowledge.

Having shown that the CLAY approach for planner synthesis is a promising research direction, we also note some of the problems that we faced while working with KIDS. To start with, the user must be reasonably familiar with the software synthesis process in order to do anything substantial with KIDS. We had to go through a steep learning curve before we could understand how to structure our theories and design our data structures to make good use of optimizations provided by KIDS. We also had to learn to keep the lay-out of the search in view while writing the distributive and monotonic laws. One open issue is whether or not it is easy for humans to give declarative control information to KIDS in the form it understands. For example, in a complex domain like the Tyre World domain, we realized that tool support is needed to specify declarative knowledge. With large number of state variable in such domains, manually encoding the control knowledge is painstaking, time-consuming and error-prone.

Despite these caveats, our overall experience demonstrates the feasibility of automating the synthesis of domain-dependent planners.

Acknowledgements: We would like to thank Doug Smith for his help with KIDS and numerous discussions on tactics, global search and distributive laws; Carla Gomes for discussing her work with us and making useful suggestions on how to write theories; and Nort Fowler for his encouragement. We also thank anonymous reviewers for their insightful comments. This research is supported by a DARPA Planning Initiative Phase 3 grant F30602-95-C-0247.

References

[1] C. Backstrom and B. Nebel. Complexity results in SAS+ planning. *Research Report, Dept Comp.and Info Sc., Linkoping Univ., Sweden*, 1993.

[2] A. Barrett and D. Weld. Partial order planning: Evaluating possible efficiency gains. *Artificial Intelligence*, 67(1), 1994.

[3] A. Blum and M. Furst. Fast planning through planning graph analysis. *Proc IJCAI-95*, pages 1636–1642, 1995.

[4] R. Fikes and N. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Readings in Planning*, 1990. Morgan Kaufmann Publ., San Mateo, CA.

[5] E. Fink and M. Veloso. Formalizing the prodigy planning algorithm. *CMU CS Tech Report CMU-CS-94-123*, 1994.

[6] C. P. Gomes. Planning in KIDS. Technical Report RL-TR-95-205, Rome Laboratory, 1995.

[7] S. Kambhampati. Refinement planning as a unifying framework for plan synthesis. *AI Magazine*, 18(2), Summer 1997.

[8] S. Kambhampati, C. Knoblock, and Q. Yang. Planning as refinement search: A unified framework for evaluating design tradeoffs in partial order planning. *Artificial Intelligence*, 76:167–238, 1995. Special Issue on Planning and Scheduling.

[9] S. Kambhampati and B. Srivastava. Universal classical planning: An algorithm for unifying state space and plan space planning approaches. *New Directions in AI Planning: EWSP 95, IOS Press, 61-75*, 1995.

[10] S. Kambhampati and B. Srivastava. Unifying classical planning approaches. Technical Report ASU CSE TR 96-006, Arizona State University, 1996.

[11] J. L. Lauriere. A language and a program for stating and solving combinatorial problems. *Artificial Intelligence*, 10:29–127, 1978.

[12] D. McAllester and D. Rosenblitt. Systematic Nonlinear Planning. *Proc. 9th NCAI-91, 634-639*, 1991.

[13] S. Minton. Automatically configuring constraint satisfaction problems: A case study. *Constraints*, 1(1), 1996.

[14] J. Penberthy and D. Weld. UCPOP: A sound, complete, partial order planner for ADL. *Proc. AAAI-94, 103-114*, 1994.

[15] S. Russell and P. Norvig. Artificial Intelligence - A Modern Approach (chap 13). *Prentice Hall, Englewood Cliffs, NJ*, 1995.

[16] D. Smith and E. Parra. Transformational approach to transportation scheduling. In *Proceedings of the 8th Knowledge-based Software Engineering Conference*, pages 14–17, 1993. Chicago, IL, Sept 1993.

[17] D. R. Smith. Structure and design of global search algorithms. *Kestrel Tech. Rep. KES.U.87.11*, 1992.

[18] B. Srivastava and S. Kambhampati. Synthesizing customized planners from specifications. *Journal of AI Research*, 1997 (to appear).

[19] M. Yoshikawa, K. Kaneko, Y. Nomura, and M. Watanabe. A constraint-based approach to high school timetabling problems: A case study. *Proc. NCAI-94*, pages 1111–1116, 1994.