Viewpoints Research Institute

Programming as Planning

Hesam Samimi

This material is based upon work supported in part by the National Science Foundation under Grant No. 0639876. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

VPRI Memo M-2009-001

Programming as Planning

Hesam Samimi
UCLA Computer Science Department
University of California, Los Angeles
Email: hesam@cs.ucla.edu

Abstract*—Inspired by the success of Planning as Satisfiability, SatPlan, we present methodologies for viewing everyday programming problems as planning problems, thus converting presumably complex programs into planning and then to satisfiability instances. We implement a planner and a solver as class modules for our target object-oriented programming language, and provide formulations for automatically encoding object-oriented message-passing style expressions into relation formulas, which can be instantiated for all possible values and asserted as clauses for the satisfiability solver. We provide our results compared to our original brute-force search planner for a simple example as well as our initial prototype for a real world example of register allocation.*

Keywords*—automated planning, planning as satisfiability, programming languages*

I. Introduction

There is a host of programming problems that are computationally hard and cannot be solved by short, simple imperative algorithms. Simple elegant programs, such as prolog relations, accomplish these tasks by relying on declarative description of problems and leaving the actual solving of problems to the underlying reasoning system using unification, search, logic and theorem proving techniques. Moreover, it is evident that neither pure theorem provers, nor blind search programs produce efficient, practical problem solvers. The unparalleled efficiency of the state-of-the-art satisfiability solvers and planners suggests that hard problems in computing are solved most efficiently using a hybrid of deductive reasoning and search. They have now become so fast that the prospect of taking the reasoning out of programs and making it part of the underlying programming language/compiler is a possibility. This project represents our initial steps in this direction.

SatPlan and a few other programs have shown that STRIPS[3] style planning problems can be encoded as satisfiability (SAT) instances and solved efficiently using SAT solvers [14]. In this report, we present our studies on presenting normal programming problems as planning instances. A programming task considered computationally difficult can be viewed as a planning problem—looking for a series of actions that satisfy a goal, and in turn as a SAT problem—essentially a series of yes/no decisions. Given the fact that is unlikely to be able to compete with the performance of a modern SAT solver in solving hard problems, this at least in theory is a noble idea. The main question, however, is how efficiently can the problem be encoded into a satisfiability instance. This report will confirm that the quality of encoding is the most important factor in making an efficient planner.

Section II describes what it means to program as planning and how it can be advantageous. In section III, we present our preliminary formulation for encoding a general programming problem into planning, and then to satisfiability instance. Section IV describes the steps involved in the implementation of putting the system together, and in section V some initial results are listed. We then conclude and list a couple of prototype examples in the Appendix sections. It should be noted that the work presented in this report is by no means complete. Our prototypes and implementations are all works in progress and unoptimized, as reflected in our results section.

II. What Is Programming as Planning

In an object-oriented programming system where problem solving is available natively to the objects in i, objects can consult the solvers to get their solutions to their computationally hard problems and then move on to do their other tasks. The problems they describe to the solver are specified as planning problems. There are a few things required in order to present a problem to a planner. An object-oriented view of a planning problem includes the class definitions with property names, procedures as actions, their variables with pre/post conditions specifications, a predicate representing the goal state, and object instances with property values representing initial state. Thus the planner is told about the available procedures—how the state can change—and it provides the planning solution that describes which procedures and when should be run in order to accomplish the goal.

A. Advantage of Programming as Planning

1) Removing complex algorithms from the code: In such a programming environment, programs can become simpler and more readable. Long, possibly inefficient code that solves hard computing tasks may be left to the underlying solver to handle.

2) Separating meanings and procedures: In programming language design, organized, readable code separates the meaning part of the program (what it is trying to accomplish*)* from the procedures part (how they accomplish it). A highly organized program becomes its own description of what it does and how it does that.

3) Heuristics and optimizations: This goal-driven look at everyday programming problems allows for a higher level of control for the program. The higher-level agent can compare the progress of the program towards the specified goal and make decisions using defined heuristics and optimizations as to how to proceed to increase the possibility of success.

B. Integrated Planning/Solving in the Programming Environment—Smart Objects

The performance of current solvers and planners is a proven fact, yet not many in the programming community use them to solve their everyday problems. It is apparent that availability of the right interface is essential. It is not practical for programmers to write wrappers and use system calls to utilize these solvers in a middle of a run. We propose that for AI techniques to be commonly used in normal programs, the solving and planning should be integrated in the programming environment. For instance, the compiler itself can consult the solver/planner to make a decision like allocation of registers. The objects themselves can be smart enough to call upon the solver just as any other method and proceed with an obtained solution.

C. Ways Objects May Consult Planners/Solver

Imagine within a compiler program it is time to allocate registers. The compiler has the program translated into its elementary form, which basically states within each instruction what variables are in their live ranges [9]. Since optimal register allocation is an NP-complete problem [10], the compiler can just describe the problem as a planning problem to the planner/solver to determine the optimum allocation choices. However, the computation never leaves the program memory space. The compiler object simply provides the necessary fields in the register allocation for the solver to read and change. Below represents our preliminary prototype for a simplistic register allocation instance; no actual results have been obtained. This prototype in its current progress is included in Appendix II.

A Register Allocation Example: *[in compiler program, need to allocate some registers...]*
Here we specify one step of allocation, which has pre- and post-conditions. The pre-condition needs to specify that the variable puzzle shape pattern must be available in the register bank. The consequence states that this area of the bank will be allotted to this variable, and the variable will be added to the `allocated-vars` for this instruction:

```
action Allocator allocate Inst Block Var consequence <post-conditions>.
rule Allocator allocate <pre-conditions>.
```

TABLE 1: Allocator Class defining an allocate action with three arguments. Pre-conditions stated separately

Here we specify the goal of register allocation, which states that at each instruction point, each variable is allocated to some register:

```
goal Allocator allocations-done try
     for every its instructions do each allocated-vars = live-vars.
```

TABLE 2: Allocator Class defining a goal: accomplished when for every instruction every variable is allocated

After the problem is specified for the planner, the compiler simply asks it to satisfy its goal:

```
MyAllocator satisfy allocations-done.
```

TABLE 3: Allocator object asking the solver to satisfy its allocations-done goal

If the problem is satisfiable, the allocations field of the Allocator object will have to right allocations for each var at each instruction and the compiler may proceed:

```
problemSolved ifTrue: [ <proceed...> ]
              ifFalse: [ <return alocationError> ]
```

TABLE 4: Allocator object inspecting the solution returned by the solver

D. Object-oriented Message-passing Style

We prefer programming languages whose expressions resemble sentences in natural language. The object-oriented message-passing style statements tend to be close to how we describe problems in natural language. Since our intention is to do planning problems within our programs, we also would like to be able to describe the planning problem in the same way. It should be noted that the choice of syntax for presenting the problem to the planner/solver is irrelevant to our main objective of doing programming as planning. The syntax in examples in the following sections is based on our syntax for JOHN system, our original brute-force search planner [13].

III. Formulations

Programming as planning requires two main encoding steps. First, given the description of a problem in a high-level object-oriented language, we need to parse the statements and encode them into relation formulas in propositional logic. These relation formulas state all information about the problem, such as what properties the objects have, how they can be changed by actions, etc. The relation formulas contain mostly variables. In the second part, we will instantiate these formulas for all combinations of values for the variables and assert them into a clausal formula in the Conjunctive Normal Form (CNF) for the SAT solver to solve. The following sections describe these steps.

A. Automatic Encoding of Problem into Propositional Logic Formulas of Relations

Program expressions involving the planning problem need to be encoded in relations, so that they can be viewed as literals for the SAT solver to assign values. For instance, an action (procedure) run at a particular time step with a particular argument values, may be encoded as a relation:

```
(obj, action, args, Time1)
```

If the solution provided by the SAT solver assigns a true value to this relation, this procedure is part of the solution plan and is run at the given time. Generally expressions describing a planning problem involve variables, instead of actual values. But relations asserted in the CNF formula for the solver can only have values. The expressions in the program are therefore initially encoded into *typed relation formulas.* The items in relations need to have the associated types so that possible values are known by enumerating all instantiated objects of such type. This is the reason only defined objects with the program are allowed as property values for any given object. This limits our flexibility for values that properties may take. Our formulations require the types of all properties to be defined. Our encoded formulas are in the form of logical implications. For asserting literals we just use implications with a true implicant. Before getting to actions (procedures in programming world), we have to define the classes and their properties:

1) Formulas encoded by class and group definitions / instantiation of objects: The first part of a planning problem description is listing objects involved in the problem. In an object-oriented environment, we have to define the classes and the properties we need to consider for each. Because we are doing planning as satisfiability, in order to use logic to find the solution we have to instantiate all relations that we know to be true, as well as false. Thus, property listings need to include the type of such properties—so that by knowing what value a particular property of an object equals, we also know which values it does not equal.

Not every property of an object is mutable. For efficiency, relations involving properties that do not change by any of the actions, need not have a time variable. We have chosen to mark the mutable properties with a star in our description; however, these can be automatically inferred by inspecting which properties are affected by any of the defined actions.

It is sometimes useful to define groups of classes (super classes). In our model of the *Hanoi2* problem, for instance, the *supporter* property of a *Disk* object can be either another *Disk*, or a *Peg* in the case of the bottom *Disk*. We defined a *Surface* group for *Disk* and *Peg* to be members of. The main advantage here is by marking the type of a property as a group, we know that the possible ground values can be the objects belonging to any of its member classes. Internally the group are just super classes of those classes, so normal rules of inheritance apply with regards to methods and properties.

Let us consider a small problem. Below we describe 2-Disk Towers of Hanoi, *Hanoi2,* example shown in Figure 1, expressed in an object-oriented, message-passing style way. This problem is solved only in 3 time steps.

```
  -A-
 --B--
======= ======= =======   Initial state
  Peg1    Peg2    Peg3

                  -A-
                 --B--
======= ======= =======   Goal state
  Peg1    Peg2    Peg3
```

FIGURE 1. Initial and Goal states in 2 disk Towers of Hanoi

Our formulation requires that all property values be defined objects; in other words, primitives like natural numbers are not allowed. This is inflexible, but it simplifies the task of encoding the problem into SAT instance since the range of possible values for all properties is known. Here is one possible model for this problem:

First we define the involved classes along with any useful grouping of classes. *Disk* and *Peg* for example, are classes, and *Surface* is a group (implemented as a super class) for the two. We have specified the type and name for several of *Peg* and *Disk* properties, such as *bigness*, *supporting*, and *supporter*. The *supporting* and *supporter* properties state the object on top of and below a given object respectively. We have marked with a star the properties that can potentially be changed by actions.

```
    group <name> <list of classes>.
    class <name> <list of property type/name pairs>.
```

TABLE 5. Syntax for defining classes and groups of classes

```
    group Surface Peg Disk.
    group Thing Disk None.
    create Towers.
    create Peg: Num bigness, * Thing supporting.
    create Disk: Num bigness, * Surface supporter, * Thing supporting.
```

TABLE 6. Examples of group and class definitions in *Hanoi2*

2) Equality: Some formulas may need to determine whether two objects of the same type are the same. While in the programming world, a simple test of *(obj1 = obj2)* would suffice; in logic domain, we need to assert relations for equality as well as non-equality of any pair of objects. Table 7 specifies a type of formula we create for class/class group definitions that encodes this relation. The relation entries are in the form of type variable pairs, with constant ones marked with a "**-**" instead of type. Generally we use the second entry as the name of the relation. The "**~**" mark implies that the negation of the relation should be asserted. When instantiating this formula, all possible combinations of the two variables *S1* and *S2*, both of type *Surface*, are inserted in the CNF formula. When the values of *S1 = S2*, this formula should not be instantiated. Later we describe how the exception cases may be handled.

```
    ~[Surface S1,- Surface-equals, Surface S2]
```

TABLE 7. Example encoded formula for object non-equality

3) Property Determinism: We need to encode relations expressing that a property of a given object could take only one value at any moment. If the property is time sensitive, the relation would include a time entry, usually in the last entry. The *Peg-supporting* formula shown in Table 8 states that any *Peg* object having a value of Thing type for its supporting property, then it must be that any other possible instances of Thing class are not the value for this property of the object.

```
    [Peg P1, - supporting, Thing T1, Time Time1]
    ==> ~[Peg P1, - supporting, Thing T2, Time Time1]
```

TABLE 8. An encoded formula for object property determinism

Now we instantiate objects for *Hanoi2* by specifying their class, name, and initial property values.

```
    make <class name> <obj name>  <list of corresponding initial property values>.
```

TABLE 9. Syntax for instantiating objects

```
    make Towers Hanoi.
    make Num Two.
    make Peg Peg1 Three B.
    make Disk B Two Peg1 A.
```
TABLE 10. Object examples in *Hanoi2*

We may define any useful relations between objects:

```
    rel-qualify <type1:var1> <name> <type2:var2> if <predicate>.
```
TABLE 11. Syntax for defining relations between two objects

```
    rel-qualify Surface:S1 smaller Surface:S2
               if ( S1 bigness = One and S2 bigness = Two ) or ...
```
TABLE 12: A relation between two *Surface* type objects in *Hanoi2*

4) Parsing general expressions into relation formulas: Object-oriented, message-passing expressions are given in right-associative <*receiver*> <*selector*> pairs. Each receiver/selector corresponds to one relation in our encoding. In a cascaded messages case such as *Peg supporting* bigness shown in Table 13, we first look up the relation *Peg-support* to get

```
[ Peg P1, - supporting, Disk D1, Time T1 ]
```

relation formula. The next message *bigness* has a receiver which is the value of *Peg supporting*, by inspecting the relation we know has a type of *Disk*. Thus the next relation in the formula should be *Disk-bigness*. The "not" in the beginning of the expression causes the second relation to be negated. The symbol "Λ" denotes logical "and."

```
    if not Peg supporting bigness = Two.
```
```
    [Peg P1, - supporting, Disk D1, Time Time1] Λ
    ~[Disk D1, - bigness, - Two ]
```
TABLE 13. An arbitrary expression and its encoded formula

5) Actions: Actions are defined by specifying arguments, consequence (post-conditions) and rules (pre-conditions). We separate action rules in separate statements. Table 14 shows the only action in the *Hanoi2* example. Here *Disk* is the actor object, and *Surface* an argument for this action. "it" and "its" are pointers to the object being described equivalent to "this" in Java, referring to the object in question. Note that the message-passing style is rather a natural way to describe the action of moving a disk from one place to another.

```
    action Disk move Surface consequence
           Surface supporting = it and
           its supporter supporting = None and
           its supporter = Surface.
```
```
    rule Disk move is its supporting = None.
    rule Disk move is Surface supporting = None.
    rule Disk move is it smaller Surface.
```
TABLE 14. *move* action in *Hanoi2*, along with its pre-conditions (rules)

We parse the expressions describing pre- and post-conditions of actions to encode the formula stating the actions, or procedures, in programming domain. An action occurring at *Time1* implies its pre-conditions at time *Time1*, and its post-conditions (consequences) at time *Time2 = Time1 + 1.*

Table 15 demonstrates formula automatically encoded by the action in our *Hanoi2* example.

```
    [ [Disk D1, - move, Surface S1, Time Time1 ] Λ
      [Disk D1,- supporter, Surface S2,Time Time1 ] ] ==>
    [ [SurfaceS1,supporter, Disk D1,Time2 ] Λ
      [SurfaceS2,supporter, - None,Time2 ] Λ
      [Disk D1,supporter,S1,Time2 ] Λ
      [Disk D1, - supporting, - None, Time Time1 ] Λ
      [Surface S1, - supporting, - None, Time Time1 ] Λ
      [Disk D1, - smaller, Surface S1 ] ]
```

TABLE 15. Encoded formula for *move* Action, implying pre- and post-conditions

Next, shown in Table 16, we describe the goal for the *Towers* object, which specifies that we like *Disk B* to be on *Peg3* and *Disk A* on *B*. Finally, we ask the solver to solve the problem (Table 17). Upon return, *A* and *B Disk* objects will be in the locations specified by the goal predicate. Appendix I includes the *Hanoi2* in its entirety.

```
    goal Towers win try B supporter = Peg3 and A supporter = B.
```

TABLE 16. Goal state in *Hanoi2*

```
    Hanoi satisfy win.
```

TABLE 17. Object asking planner to satisfy its goal

We have described a possible syntax and semantics for object-oriented description of a planning problem. The more difficult next step is to encode general expressions into relations. However, this is logically not enough. The *frame problem* [7] deals with how to logically specify the frame axiom: all relations in previous time unaffected by an undertaken action hold in the time step after the action happens. The consequence clause of the action only generates relations for the affected properties at time Time2, but says nothing about all the ones that are not changed.

Frame Axiom: To handle the frame problem we have to define more formulas for each action. Note that we only need to consider mutable properties of objects.
- For those mutable properties that the action doesn't involve, we assert two cases: if the property/value relation is true at *Time1,* then it is true at *Time2*; and if false at *Time1*, it will also be false at *Time2*.
- For those properties the action does affect, we have to assert formulas specifying that provided object is not equal to the objects in the action post-conditions using the "equality relations" we asserted before, their property values propagate from previous time.

```
    [ [Disk D1, - move, Surface S1, Time Time1 ] Λ
      [Disk D1,- supporter, Surface S2,Time Time1 ] Λ
     ~[Surface S0, - Surface-equals, Surface S1 ] Λ
     ~[Surface S0, - Surface-equals, Surface S2 ] Λ
      [Surface S0, - supporting, Thing T1, Time Time1 ] ] ==>
    [Surface S0, - supporting, Thing T1, Time Time2 ]
```

```
    [ [Disk D1, - move, Surface S1, Time Time1 ] Λ
      [Disk D1,- supporter, Surface S2,Time Time1 ] Λ
     ~[Surface S0, - Surface-equals, Surface S1 ] Λ
     ~[Surface S0, - Surface-equals, Surface S2 ] Λ
     ~[Surface S0, - supporting, Thing T1, Time Time1 ] ]==>
    ~[Surface S0, - supporting, Thing T1, Time Time2 ]
```

TABLE 18. Formulas expressing the Frame Axiom—two cases for property relation true and false

Action Exclusivity: Planning problems work with the premise that with every action the time step is moved forward by one unit. Thus only one action at any point of time is possible, and we need a formula that asserts this fact. We have to assert that only one set of action/argument values is possible at any time. Table 18 lists the generated formula for *Disk* action in *Hanoi2* problem. The only instantiation exception combination for this formula is when *D1 = D2* and *S1 = S2*. Note that this formula would not suffice if more than one type of action is possible in a problem. In such a case, a more general formula is needed specifying that any action by any object negates the occurrence of any other action by any other object.

```
    [Disk D1, - move, Surface S1, Time Time1 ]
    ==> ~[Disk D2, - move, Surface S2, Time Time1 ]
```

TABLE 19. Sample formulas generated for Action Exclusivity

B. Instantiation of Propositional Logic Formulas

In order to use relations as literals for the SAT solver, we need to instantiate relation formulas for all possible combinations of values for the variables. Since the variables are marked with their types, the possible values for each variable are known by enumerating all instances of each type. Some variables may need to be preset for some formulas. For instance, when instantiating the relation formula encoding an action at time 0, then the *Time1* and *Time2* variables given in the formula of Table 15 need to be preset to values 0 and 1 respectively.

The number of possibilities of instantiation of formulas with multiple variables can be large. For example, the encoded formula of Table 15 has variables *D1*, *S1*, and *S2* besides the *Time* variables. Hence if there are *d Disks* and *s Surfaces (Disks + Pegs)* in the Hanoi problem, we have *d * s * s* possible combination of values. In general, encoding relation formulas with multiple variables is commonly avoided in implementation of planning as satisfiability by Operator Splitting [4], effectively breaking action relations into multiple single-variable ones. Our preliminary formulations of automatic encoding do not consider any optimizations such as this.

Given the inefficient encoding, we implemented instantiation and assertion of formulas as function calls in order to reduce the instantiation time. When parsing an expression and defining the corresponding logical relation formula, an instantiation function for the formula for the class involved is generated dynamically. This function takes the formula variables as arguments and to bind the variables in relations with arguments and produce a relation formula with only values, then asserts the formula as a clause in the solver's CNF.

The instantiation/assertion function for each formula would include any combination values to exclude in assertions. For example in the *Surface-equals* formula in Table 7, the only instantiation value pair that must not be asserted is when *S1 = S2*. In that case, then non-equality does not hold.

```
    (define-send 'Surface-not-equals <variables>
      (if (== S1 S2)
            [ignore ...]
            [otherwise: generate relations of the formula type given values and
              assert into cnf]))
```

TABLE 20. A dynamically generated Instantiation Function for *Surface-not-equals* formula handling an instantiation exception case

Formula categories: It will be shown that in our planning algorithm, we do not instantiate all formulas the same point. For this reason, we specified 3 categories for the relation formulas. For relations that are time independent we assign a category 0, and only instantiate them once in the beginning. Those relations having to do with current "time", such as the *Peg-supporting* formula, have a category of 1. These formula relations need to be instantiated for each time step iteration in the planning algorithm. In section IV.B we explain the *SatPlan* algorithm that determines the value of Time variable at each iteration. Finally, relations associated with actions are assigned a category of 2. They are used to determine the relations at the next time step, rather than the current time. This category will only be asserted at the end of each planning iteration, just before starting a new iteration with an incremented time.

IV. Implementation

In order to do programming as planning and planning as satisfiability natively, the programming language needs to implement both the planner and SAT Solver as library classes available to any object. Our target programming language, *COLA* [12], which offers Smalltalk style high-level object-oriented message-passing language, as well as Lisp style, functional programming with dynamic code generation [11]. The object-oriented part of COLA, *Pepsi*, contains the classes as modules which can be used both in functional and objective programs. Hence Pepsi is the target language for our Planner and satisfiability solver. We implemented *MiniSat* [2] a compact state-of-the-art SAT solver, and for Planning, we used our own prototype with a lot of similarities with SatPlan which does planning as satisfiability.

A. Satisfiability Solver Implementation

We have implemented MiniSat as class for COLA family of languages. This implementation was based on the C++ MiniSat version [2]. The two common methods used with the class are *addClause* and solve.

`[ MiniSat addClause: clause ]` *- adds a clause, doing simplifications and propagations if possible*
`[ MiniSat solve ]` *- solves the CNF, returns if satisfiable: true,* `[MiniSat model ]` *has the solution else: false*

TABLE 21. MiniSat class and *addClause* and *solve* methods

Implementing a low-level C++ program into a high-level object-oriented language was not trivial. In our preliminary implementation of MiniSat, we overlooked a lot of efficiency design choices such as bit operations and clever data structures that has made it a competitor solver. We neglected such design choices in the interest of time and implemented many data structures in the most obvious way. In summary, our preliminary implementation has been used as a prototype and cannot perform comparably with the original C++ until optimized.

B. Planner Implementation

Our planner is implemented on top of functional core of COLA, also known as *Jolt*. The objects and properties involved in the planning problem are in fact objects and messages for such objects in the language. The actions and goal searches are just methods run by those objects. The advantages are obvious. Everything is supported by the underlying interpreter: variable bindings, inheritance, supported data structures like lists and sets, message sending, etc. This is also a requirement since our goal is to have any program use the planning/solving whenever needed, as part of own library.

1) A model for relations: In order to do Planning as satisfiability, we had to store objects and their fields as relations. We developed a model of relations and relation formulas as a Pepsi class for this purpose that supports definition of typed relations, formulas of such relations as implications, and instantiation and assertion of such formulas as clauses for the SAT solver. Table 22 lists just a few methods used with the *RelationModel* class, used extensively in our planner implementation.

```
[ RelationBase defineRelation: <relation name>
                         type: <list of type names> ]
```
- define a relation

```
[ RelationBase defineFormula: <formula name>
                impliedsType: <list of implieds relation names>
              implicantsType: <list of implicants relation names>
                    implieds: <list of implieds relation templates>
                  implicants: <list of implicants templates>
             isBidirectional: <boolean> ]
```
- define a new propositional logic formula between relations

TABLE 22. RelationModel class and two of its methods

There are also methods for instantiation relations and formulas, as well as asserting them in a CNF formula as clauses for the SAT Solver.

2) Planning algorithm: We follow the planning as satisfiability algorithm as in the SatPlan planner program. Table 23 restates the algorithm here for convenience. Category 0 formulas are the time independent clauses (those unaffected by any actions), Category 1 are time sensitive ones, and Category 2 are action specifications.

```
currTime := 0.
instantiate and assert formulas of category 0.
( currTim < TimeoutTime ) WhileTrue: [
    instantiate and assert relation formulas of category 1, setting Time1 = currTime.
    assert goal is satisfied at currTime.
    ( SATSolver solve )
         ifTrue: [ return the extracted plan from model ]
        ifFalse: [ instantiate and assert relation formulas of category 2,
                       setting Time1 = currTime and Time2 = currTime+1.
                   assert disjunction of all possible actions at currTime. ].
    currTime := currTime + 1.
]
return false.
```

TABLE 23. SatPlan Algorithm

Starting from current time of 0, the goal is asserted and checked for satisfiability. If not satisfied, then we assert that one of many possible actions is done at time 0 and then repeat the process to check the satisfiability at time 1, and so on.

V. Results

Given the fact that our preliminary implementations were at the prototype level and not optimized, our results were very promising. As expected, when comparing small problems such as Tower of Hanoi problem with 2 Disks, our planning as satisfiability implementation performs much worse than our original method of brute-force search due to the overhead of encoding relations. The *Hanoi2* problem is satisfiable in only 3 time steps, and searching all possibilities takes much less than a second. With planning as satisfiability methods, there is all the overhead of instantiation of all possible relations for times between 0 and 3 and feeding the resulting CNF to the solver.

However, as we test larger problem instances, the overhead of encoding relations becomes less significant compared to exponential growth of the search tree in the blind search case. For instance, our original brute-force search planner on a Hanoi4 instance, did not finish after 2 hours without finding the 15-step solution plan. Even with our inefficient encodings and solver, the problem took about an hour to encode into a CNF, and 30 minutes with our MiniSat implementation to solve. These results indicate that solving general problems into satisfiability instances is in fact a right decision. While the *Hanoi4* problem was not solved in 2 hours using brute-force search, the C++ MiniSat solved the equivalent CNF version of the problem in 9 seconds!

It should be noted that the while growing, a problem blows up exponentially with brute-force search, the encoding time for every new time step just changes linearly. Also the solving time for the SAT solver does not increase exponentially in the extent of the search methods. The results also show that most of the run time is spent on instantiation of encoded formulas for all considered time steps. The time spent by the MiniSat is much less than encoding time. It is evident that the quality of the planner mostly depends on how efficient the encoding process can be.

Problem	JOHN Total Time	JOHN + SatPlan Total Time	Encoding Time	# Vars	# Clauses	Our MiniSat Time	C++ MiniSat Time
Hanoi2	*< 1s*	*24s*	*22s*	*250*	*5K*	*2s*	*< 1s*
Hanoi3	*12s*	*5min*	*3.5min*	*630*	*53K*	*1.5min*	*< 1s*
Hanoi4	*unsolved > 2hr*	*1.5hr*	*55min*	*1.6K*	*345K*	*34min*	*9s*

TABLE 24. Results comparing JOHN, our original brute-force search planner, JOHN SatPlan implementation (JOHN+SatPlan), Encoding Time, JOHN MiniSat implementation (JOHN+MiniSat), and C++ MiniSat on Hanoi instances of sizes 2-4

VI. In Progress and Future Work

As noted, all our results and prototypes represent work in progress. Other than the necessary efficiency optimizations, like redesigning data structures and fixing existing bugs, in order to make programming as planning a reality we need to work on many fronts, some of which are listed below.

1) Reducing complexity of encodings: *Operator Splitting*—Modern planners break relations into single variable relations [4]. This reduces the literals needed to represent all relations as well as number of instantiations. For instance, The *Disk-move* action would instead be encoded as separate relations `Disk-move-actor [Disk D1, Time T1 ]`, `Disk-move-var [Surface S1, Time T1 ]`, etc.

2) Optimizing MiniSat implementation: In the interest of time, we overlooked clever bit computations done by MiniSat C++ version and implemented most convenient data structures. This has a major efficiency hit for our implementations and the run times may be an order of magnitude worse. For instance, a lot of computation is done by clever bit operation like shifts and masks for efficiency purposes. Also, MiniSat uses an Ordered Heap structure to remove the variable with the minimum order for assigning variables, which is a constant time operation.

3) Prototype for a real world programming example: Appendix II represents our initial prototype for The Register Allocation problem. This represents a simplified instance of register allocation as puzzle solving [9] and is currently under test. This is promising since it is possible to describe the problem in less than a hundred lines and have the solver provide the solution results in faster code with much smaller, cleaner and more readable code.

4) Delivering the planner/solver package as a class usable by any object: Our current work requires programs to use the syntax of our planner to specify problems. While we have implemented our Solver as a class module, our SatPlan implementation is written in functional part of the language, which means programs can only run on top of it. In order for our planner/solver package to be truly usable by any program in the programming language environment, the planner itself also needs to be rewritten as a class module. This is not a big task.

5) Incorporating heuristics and optimizations into problem: It has been shown that domain specific heuristics can be incorporated in the CNF generated for the SAT solver in a planning problem [5]. Our brute-force planner supports heuristics and optimizations, which has a huge impact on the solution time of a given problem. Heuristics can be defined for reordering search tree branches that make reaching the goal state faster. Optimizations can also be defined that state which actions at any given point are most appropriate to explore. Encoding such heuristics in the encoded relations usable by the SAT solver isn't a trivial task. We would like to study ways for this and also provide interesting prototypes.

VII. Related Work

Planning as Satisfiability brought in 1992 by Kautz et al made it evident that large problems can be translated efficiently into propositional satisfiability problems. Many improvements have been studied over the years. GraphPlan [1] makes use of graph theory to achieve efficient planning. BlackBox [6] is a variant of SatPlan that employs GraphPlan algorithms. There has been a lot of work on top of these planners.

Another prominent type of planners is based on Probabilistic and Bayesian Reasoning, and uses techniques such as Belief Propagation to do planning [17].

On the programming languages side, most of the research has been focused on applying automated reasoning techniques for a specific application such as general compiler Optimizations, register allocation [9], and debugging [8]. John Whaley et al use Binary Decision Diagrams to Using Datalog programs and and Binary Decision Diagrams (BDDs) for different types of Program Analyses such as pointer analysis and race condition [8].

VIII. Conclusion

We have presented our preliminary studies on bringing the power of satisfiability solvers to the programming languages. We discussed ways objects in a programming language may avoid difficult computational tasks, such as NP-complete problems, by calling on a planner to solve them. The problem can be described in a high-level syntax compatible with the language itself. We then enumerated our formulations for how the planner can encode different type expressions into propositional logic formulas. The planner will then instantiate these formulas to obtain relations that can be given to the SAT solver as clauses. A few optimizations for encoding steps, such as dynamic generation of instantiation functions, were proposed.

For the most part, all our implementations—building relations, encodings, MiniSat solver—need to be redesigned and optimized for performance. In the results section, while the brute-force search planner did not finish solving the *Hanoi4* problem even in 2 hours, MiniSat solved the clausal equivalent of the problem in 9 seconds! This outstanding result inspires us to work on the efficiency of our encodings, in an effort to make general programming as planning a possibility.

Acknowledgements

We thank Kathryn Hewitt for revising this report.

References

[1] A. Blum and M. Furst, *Fast planning through planning graph analysis*, In Artificial Intelligence, 90:281-300, 1997.

[2] N. Eén and N. Sörensson, *An extensible SAT-solver*, In Proc. of the Sixth International Conference on Theory and transformation of C programs, extended version 1.2.

[3] R. Fikes and N. Nilsson, *Strips: A new approach to the application of theorem proving to problem solving*, Artificial Intelligence, Vol. 2, No. 3-4. (1971), pp. 189-208.

[4] H. Kautz, D. McAllester, and B. Selman, *Encoding plans in propositional logic*, Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR-96), Boston, MA, 1996.

[5] H. Kautz, and B. Selman, *The Role of Domain-Specific Knowledge in the Planning as Satisfiability Framework*, Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems, 181-189. Menlo Park, Calif.: AAAI Press.

[6] H. Kautz and B. Selman, *Unifying SAT-based and graph-based planning*, Proc. IJCAI-99, Stockholm.

[7] H. Kautz and B. Selman, *Pushing the envelope: Planning, propositional logic, and stochastic search*, Proceedings AAAI-96.

[8] M. Lam, J. Whaley, B. Livshits, M. Martin, D. Avots, M. Carbin, and C. Unkel, *Context-sensitive program analysis as database queries*, Proceedings of the 24th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'05), pages 1-12, 2005.

[9] F. Pereira and J. Palsberg, *Register allocation by puzzle solving*, 2008, http://compilers.cs.ucla.edu/fernando/projects/puzzles/.

[10] F. Pereira and J. Palsberg, *Register allocation after classic SSA elimination is NP-complete*, In FOSSACS, pages 79-93, Springer, 2006.

[11] I. Piumarta, *Accessible Language-Based Environments of Recursive Theories*, VPRI Research Note, RN-2006-001-a.

[12] I. Piumarta and A. Warth, *Open, reusable object models*, S3, 2008.

[13] H. Samimi, *JOHN—A Knowledge Representation Language*, http://www.advicetaker.org/pmwiki/pmwiki.php?n=Main.AnOverview.

[14] B. Selman and H. Kautz, *Planning as satisfiability*, In Proceedings ECAI-92, pages 359-363, 1992.

IX. Appendix

In the following pages we provide two sample planning programs written in *JOHN*[13] syntax. Appendix I contains the *Hanoi4* program, along with some of the inferred formula encodings by the planner. Appendix II represents our very preliminary attempt at modeling a simple register allocation program. The model is based on *Register allocation by Puzzle Solving* [9]. No results have been obtained yet. We have included this section as we believe a fairly small code, with a high performance underlying reasoning engine, can make an elegant efficient allocator program for a compiler.

Appendix I: Hanoi4 Program

```
    -A-
   --B--
  ---C---
 ----D----
=========== =========== ===========   Initial state
    Peg1       Peg2        Peg3

                            -A-
                           --B--
                          ---C---
                         ----D----
=========== =========== ===========   Goal state
    Peg1       Peg2        Peg3
```

FIGURE 2. Initial and Goal states in 4 disk Towers of Hanoi

TABLE 25. *Hanoi4*: Towers of Hanoi with 4 Disks, along with some generated formula encoding

```
start-world towers-of-hanoi.

;; GROUPS
group Surface Peg Disk.
group Thing NotApplicable Disk.

;; CLASSES
create NotApplicable.
create Num.
create Towers.
create Peg: Num bigness, * Thing supporting.
create Disk: Num bigness, * Surface supporter, * Thing supporting.

;; OBJECTS
make NotApplicable None.
make Num One. make Num Two. make Num Three. make Num Four.make Num Five.
make Towers Hanoi.
make Peg Peg1 Five B. make Peg Peg2 Five None. make Peg Peg3 Five None.
make Disk D Four Peg1 C.
make Disk C Three D B.
make Disk B Two C A.
make Disk A One B None.

;; QUALIFFICATIONS (RELATIONS)
rel-qualify Surface:S1 smaller Surface:S2 if
           ( S1 bigness = One and S2 bigness = Two ) or
          ( S1 bigness = One and S2 bigness = Three ) or
          ( S1 bigness = One and S2 bigness = Four ) or
          ( S1 bigness = One and S2 bigness = Five ) or
          ( S1 bigness = Two and S2 bigness = Three ) or
          ( S1 bigness = Two and S2 bigness = Four ) or
          ( S1 bigness = Two and S2 bigness = Five ) or
          ( S1 bigness = Three and S2 bigness = Four ) or
          ( S1 bigness = Three and S2 bigness = Five ) or
          ( S1 bigness = Four and S2 bigness = Five ).

rel-qualify not Surface:S1 smaller Surface:S2 if S2 smaller S1.

;; ACTIONS

action Disk move Surface consequence
       Surface supporting = it and
       its supporter supporting = None and
       its supporter = Surface.
rule Disk move is its supporting = None.
rule Disk move is Surface supporting = None.
rule Disk move is it smaller Surface.

;; GOALS
goal Towers win try B supporter = Peg3 and A supporter = B.

;; QUERY
Hanoi satisfy win.
end-world.
```

Appendix II: A Simplistic in-Progress Register Allocation Prototype Based on 'Register Allocation By Puzzle Solving' [9]

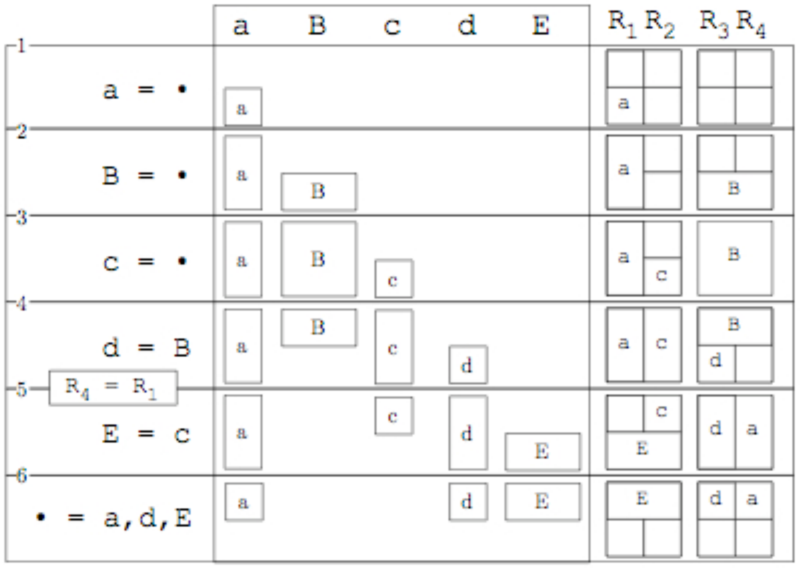

FIGURE 3. An example program (left). Puzzle pieces (center). Register assignment (right). (Figure from: Register Allocation By Puzzle Solving [9])

```
         R1  R2  R3  R4
         -------------
  Row1  |0  |2  |4  |6  |  1
         -------------
  Row2  |1  |3  |5  |7  |  2
         -------------
           1   2   3   4

  -----            ---
 |     |  -----   |   |  ---
 | P22 | | P12 | |P21| |P11|
  -----   -----   ---   ---
```

FIGURE 4. Model for Register Bank per Instruction (top). Register Blocks as Puzzle Pieces (bottom)

TABLE 26. A Simplistic Register Allocation Prototype, By Puzzle Solving [9]

```
start-world ssa.
create Allocator instructions registers.
create Instruction id live-vars allocated-vars free-blocks allocations.
create Block row reg index.
create Reg num.
create Row num.
create Var puzzle-piece half-puzzle-piece start-inst end-inst.
create PuzzlePiece x-length y-length.
qualify Block area-span-x: X area-span-y: Y
select all Block by which row num <= its row num and which row num > (its row num
- X) and
which reg num >= its reg num and which reg num < (its reg num + Y).
qualify Block span-for-var: Var at: Inst
its area-span-x: ((Var inst-piece: Inst) x-length)
area-span-y: ((Var inst-piece: Inst) y-length).
qualify Var inst-piece: Inst its puzzle-piece
if not (Inst = its start-inst or Inst = its end-inst).
qualify Var inst-piece: InstId its half-puzzle-piece.
qualify List contains: L1 is for every L1 do each in it.
make PuzzlePiece P22 2 2.
make PuzzlePiece P12 2 1.
make PuzzlePiece P21 1 2.
make PuzzlePiece P11 1 1.
make Reg R1 1. make Reg R2 2. make Reg R3 3. make Reg R4 4.
make Row Row1 1. make Row Row2 2.
make Var Va P21 P11 I1 I6.
make Var VB P22 P12 I2 I4.
make Var Vc P21 P11 I3 I5.
make Var Vd P21 P11 I4 I6.
make Var VE P22 P12 I5 I6.
make Block B11 Row1 R1 0. make Block B12 Row2 R1 1.
make Block B21 Row1 R2 2. make Block B22 Row2 R2 3.
make Block B31 Row1 R3 4. make Block B32 Row2 R3 5.
make Block B41 Row1 R4 6. make Block B42 Row2 R4 7.
make Instruction I1 1 [ Va ] [ ] (all Block) [ ].
make Instruction I2 2 [ Va, VB ] [ ] (all Block) [ ].
make Instruction I3 3 [ Va, VB, Vc ] [ ] (all Block) [ ].
make Instruction I4 4 [ Va, VB, Vc, Vd ] [ ] (all Block) [ ].
make Instruction I5 5 [ Va, Vc, Vd, VE ] [ ] (all Block) [ ].
make Instruction I6 6 [ Va, Vd, VE ] [ ] (all Block) [ ].
make Allocator Al (all Instruction) (all Reg).
action Allocator allocate Inst Block Var consequence
      Inst free-blocks = Inst free-blocks - (Block span-for-var: Var at: Inst) and
      Inst allocated-vars = Inst allocated-vars + Var and
      Inst allocations = Inst allocations + [ [ Block, Var ] ].
rule Allocator allocate is ((Inst free-blocks) contains: (Block span-for-var: Var
at: Inst)) = yes.
goal Allocator done try for every its instructions do each allocated-vars size =
live-vars size.
end-world.
```