Finding Parallel Regions with Temporal Planning

Claudio Scheer
Master's Degree in Computer Science
Pontifical Catholic University of Rio Grande do Sul - PUCRS
Porto Alegre - RS, Brazil
claudio.scheer@edu.pucrs.br

Abstract

Abstract.

Introduction.

Bibliography

Temporal Plannign

According to (Haslum et al. 2019), actions in temporal planning have a duration. Therefore, the planner will try to find a schedule in which some actions can be executed in parallel.

There are different approaches that can be used to formalize temporal actions with PDDL. In this paper, I used `:durative-action`. This action is represented in four sections, as listed below.

- `:parameters`: parameters needed to execute the action;
- `:duration`: time the action takes to run;
- `:condition`: conditions that need to be respected to apply the effects;
- `:effect`: effects that will be applied to the state;

The sections `:condition` and `:effect` are separated in three categories: `at start`, `over all` and `at end`. As described by (Haslum et al. 2019), these categories represent the conditions and effects used at each stage of the action. The `at start` statements are used when starting the action. The `over all` statements are used during the time the action is being executed. The `at end` statements are used at the end of the action.

The following section may not be necessary.

I must mention that `:durative-action` can be translated into instantaneous actions. Here is the paper: (Scala et al. 2016).

STP

I used the STP (Simultaneous Temporal Planner) planner, introduced by (Blanco et al. 2018), to find the best temporal plan. STP uses a modified version of the Fast Downward (Helmert 2006) planner that can generate a temporal plan. The version used in this paper is provided by the Artificial Intelligence and Machine Learning Group - Universitat Pompeu Fabra[1].

The STP planner needs to receive as a parameter the maximum number of actions that can be executed at the same time. When finding parallel regions, this parameter is a problem, because we do not know how many instructions can be executed in parallel. Therefore, in some cases, it is necessary to test different values for this paramenter.

The higher this number, the more time it takes to solve the problem.

Source Code Translation

Describe here some patterns on how to convert source code into the PDDL problem.

Formalization

I used only two `:durative-action` to formalize the problem. One to handle assignment instructions and another to handle binary operations. Binary operations must be understood as any operations (sum, multiplication, XOR, etc) that involve two variables.

All `:durative-action` must have a duration time. However, since my objective is not to cover the execution time of the instructions in this paper, I set the duration time to 1 for all actions. This approach ensures that when operations cannot be executed in parallel, the total execution time will be increased by one. In a future work, it may be a good approach to define different duration time for each instruction.

In a future work, it may be a good approach to define different duration time for each instruction.

By definition, each operation and assignment instruction must have an identifier. With this identification, we can create the dependency tree for the instructions. The dependency tree makes it easy to test whether all dependencies for a specific instruction have already been executed. Since each problem has a specific dependency tree, the dependency tree must be defined in the initial state of the problem. In the initial state, in addition to the dependency tree, the identifier for the operation and assignment instructions must also be defined.

[1]https://github.com/aig-upf

As discussed in Section Temporal Plannign, the sections `:conditions` and `:effects` have three different categories. In my formalization, all preconditions must be respected at the beginning of the action and the effects is applied only at the end of the action.

Assignment action

The assignment action will receive the assignment instruction and the instruction identifier as a parameter. The first two conditions will ensure that the identifier belongs to the the instruction and that the assignment has not yet been executed.

The third condition is common the all durative actions. The `forall` operator loops through all identifiers in the dependency tree, testing whether the identifier is parent of the current instruction and whether the parent instruction have already been executed.

The Table 1 shows the truth table that represents the condition within the `forall` operator. The variable A represents that the current identifier is a child of the parent identifier. B represents that the parent identifier has already been executed. As shown in the truth table, the only case where the condition returns false is when the current identifier is a child of the parent identifier and the instruction represented by the parent identifier has not yet been executed.

A	B	$\neg A \vee B$
1	1	1
1	0	0
0	1	1
0	0	1

Table 1: Dependency tree condition truth table

The `:effect` of the assignment action is to mark the instructions as executed. The Listing 1 shows the formalization of the assignment action.

```
(:durative-action assignment
   :parameters (
      ?instruction_id - id
      ?id - assignment
   )
   :duration (= ?duration 1)
   :condition (and
      (at start (assignment_id ?id ?instruction_id))
      (at start (not (executed_assignment ?id)))
      (at start (forall (?parent - id)
         (or
            (not (dependency_tree ?parent ?instruction_id))
            (executed_instruction ?parent)
         )
      ))
   )
   :effect (and
      (at end (executed_instruction ?instruction_id))
      (at end (executed_assignment ?id))
   )
)
```

Listing 1: Formalization of the assignment action

Binary operation action

This action is responsible for executing a binary operation between two variables. The action takes as parameters the two assignment variables that will be used in the operation and the assignment instruction that will receive the result of the binary operation.

Here, the conditions to execute the action are that the binary operation has not yet been executed and the assignment instructions of the two input variables have already been executed.

As shown in Listing 2, the effects are similar to the previous action: the binary operation is marked as executed, releasing the next instructions that holds in this operation.

```
(:durative-action binary_operation
   :parameters (
      ?instruction_id - id
      ?idA - assignment
      ?idB - assignment
      ?operation_id - operation
      ?idC - assignment
   )
   :duration (= ?duration 1)
   :condition (and
      (at start (operation_id ?operation_id ?instruction_id))
      (at start (forall (?parent - id)
         (or
            (not (dependency_tree ?parent ?instruction_id))
            (executed_instruction ?parent)
         )
      ))
      (at start (not (executed_operation ?operation_id)))
      (at start (not (executed_binary_operation ?idA ?idB ?
          operation_id ?idC)))
      (at start (executed_assignment ?idA))
      (at start (executed_assignment ?idB))
   )
   :effect (and
      (at end (executed_instruction ?instruction_id))
      (at end (executed_operation ?operation_id))
      (at end (executed_binary_operation ?idA ?idB ?operation_id ?
          idC))
   )
)
```

Listing 2: Formalization of the binary operation action

Results

In this section, I will show a simple example of formalization. Regarding our objective, we hope that the temporal plan found by the planner shows the instructions that we can execute in parallel.

The example is represented by the source code shown in Listing 3. This simple source code sums two variables and set the result in the variable `c`. The variables `a` and `b` are not dependent on each other. Therefore, we must be able to execute the two assignment instructions in parallel, as shown in Figure 1.

```
#include <iostream>

int main()
{
   int a = 3;
   int b = 3;
   int c = a + b;
   return 0;
}
```

Listing 3: Source code

The formalization of this problem is as shown in Listing 4.

```
(:init
   (executed_instruction id0)

   (assignment_id assignmentA id1)
   (assignment_id assignmentB id2)
   (operation_id sumAB id3)
   (assignment_id assignmentC id4)

```

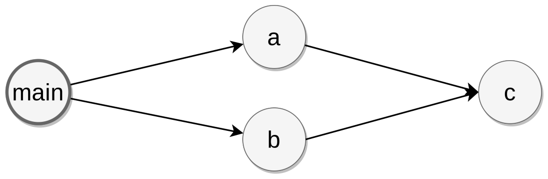
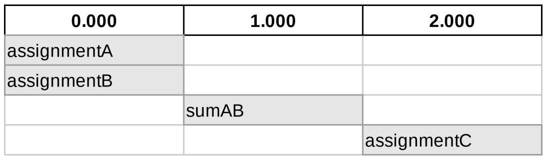


Figure 1: Dependency tree

```
 9     (dependency_tree id0 id1)
10     (dependency_tree id0 id2)
11     (dependency_tree id1 id3)
12     (dependency_tree id2 id3)
13     (dependency_tree id3 id4)
14  )
15
16  (:goal (and
17     (executed_assignment assignmentA)
18     (executed_assignment assignmentB)
19     (executed_binary_operation assignmentA assignmentB sumAB
            assignmentC)
20     (executed_assignment assignmentC)
21  ))
```

Listing 4: Source code formalization

The main point of the Listing 4 is on lines 9 and 10. In these lines, I define the `a` and `b` instructions as dependent only on the main instruction[2]. The STP output of this formalization is represented visually in Figure 2.

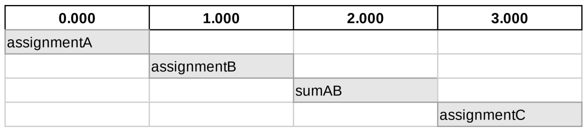


0.000	1.000	2.000
assignmentA		
assignmentB		
	sumAB	
		assignmentC

Figure 2: STP parallel plan

The Figure 2 shows that, as soon as the program starts, we can execute the `assignmentA` and `assignmentB` instructions at the same time. To execute the `sumAB` instruction, we must wait until the previous instructions are executed. Finally, to execute the `assignmentC` instruction, all previous instructions must be executed.

Imagining a source code similar to Listing 3, where the variable `b` is a result of the variable `a` plus 1, we would have a different dependency tree. Now, instead of line 10 in Listing 4, would be replaced by `(dependency_tree id1 id2)`, ensuring that variable `b` is assigned only after the variable `a`. The output of this second example is shown in Figure 3.



0.000	1.000	2.000	3.000
assignmentA			
	assignmentB		
		sumAB	
			assignmentC

Figure 3: STP non-parallel plan

[2]The main instruction represents the starting point of the application.

Conclusion

References

[Blanco et al. 2018] Blanco, D. F.; Jonsson, A.; Verdes, H. L. P.; and Jimenez, S. 2018. Forward-search temporal planning with simultaneous events.

[Haslum et al. 2019] Haslum, P.; Lipovetzky, N.; Magazzeni, D.; and Muise, C. 2019. *An Introduction to the Planning Domain Definition Language*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers.

[Helmert 2006] Helmert, M. 2006. The fast downward planning system. *Journal of Artificial Intelligence Research* 26:191–246.

[Scala et al. 2016] Scala, E.; Haslum, P.; Thiébaux, S.; and Ramírez, M. 2016. Interval-based relaxation for general numeric planning. In Kaminka, G. A.; Fox, M.; Bouquet, P.; Hüllermeier, E.; Dignum, V.; Dignum, F.; and van Harmelen, F., eds., *ECAI 2016 - 22nd European Conference on Artificial Intelligence, 29 August-2 September 2016, The Hague, The Netherlands - Including Prestigious Applications of Artificial Intelligence (PAIS 2016)*, volume 285 of *Frontiers in Artificial Intelligence and Applications*, 655–663. IOS Press.