Intelligent Robotic Systems
A.A. 2018 - 2019

ScalaPlan

Marco Brighi
marco.brighi4@studio.unibo.it

June 10, 2019

Contents

Abstract

The main objective of this report is to show the design and the implementation of ScalaPlan, a simple application of planning. A planning application is a specific software that aims to plan a sequence of action in order to reach a goal, considering an initial situation. Browsing through literature, it is possible to find many ways to resolve a planning problem. This specific application contains a SAT-based resolver. That means that the planning problem is turned into a SAT problem and solved. The first chapter describes the role and the functionalities requested by the application. It gives you also a synthetic explanation of what it means planning and what we expect from it. The second chapter shows the design architecture of the solution according to the principles of the Software Engineering. The third chapter describes some interesting details of the implementation made using the Scala programming language. It also details the libraries included in the project. At the end, the last chapter encloses the conclusions and possible future developments. The result obtained is part of the project made for the Intelligent Robotic Systems exam.

Chapter 1

Requirements Analysis

The main objective of this work is to create a planning application. Provided a description of the initial state, a planning application looks for a sequence of action in order to reach a specific goal. From this simple description we can understand that there are three main entities in order to describe the domain: initial state, goal and actions. Thanks to this representation, a solver can find a consistent plan containing the actions for reaching the goal. The application must be able to be used directly by code or by using a user-friendly graphical interface.

1.1 Description

The formalism used to describes these three entities is one the crucial choices to be made. The first and most important language created to describe states and actions is *STRIPS (Stanford Research Institute Problem Solver)*. It was created in order to program the robot *Shakey* and includes even a specific algorithm of resolution. The adoption of this language is required in this project in order to obtain a standard STRIPS-like description of the domain. STRIPS describes the initial state and the goal very simply: a conjunction of atomic formulas. Thanks to the *closed world assumption* everything that is not directly defined as true is false. The actions are instead uniquely identified thanks to their name and describe with three list of fluents:

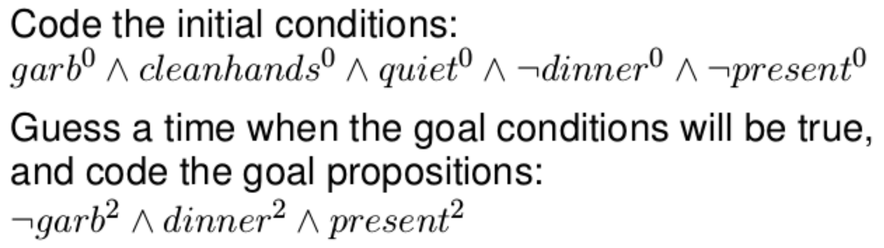
- prerequisite list, it contains the fluents that must be verified to allow the execution of the action;
- add list, it contains the fluents that must be added in the successor state after the execution of the action;

- delete list, it contains the fluents that must be removed from the current state.

1.2 Solver

Naturally, another essential component for the application is the solver, the engine that aims to produce a consistent plan from a complete description. In literature we can find different types of solvers, each one with advantages and disadvantages. For this particular project the use of a SAT-based solver is required.
A SAT-based solver is nothing but a classic solver of SAT, a deeply studied NP-complete problem of determining if there exists an interpretation that satisfies a given boolean formula. In other words, it asks whether the variables of a given boolean formula can be consistently replaced by the values true or false in such a way that the formula evaluates to true. If this is the case, the formula is called satisfiable.
A planning problem can be easily translated into a SAT problem. This is possible following a set of translation rules:

- every proposition or action must be accompanied by an index describing the time to which must be associated; the zero is associated at the initial state by definition; these strategy aims to bring the time dimension into the SAT problem definition;
- the initial state and the goal are coded as conjunction of fluents; naturally the goal's propositions are associated with the last index that also represents the maximum duration allowed for the plan;



Code the initial conditions:

$garb^0 \wedge cleanhands^0 \wedge quiet^0 \wedge \neg dinner^0 \wedge \neg present^0$

Guess a time when the goal conditions will be true, and code the goal propositions:

$\neg garb^2 \wedge dinner^2 \wedge present^2$

Figure 1.1: An example of initial state and goal.

- for each fluent a *successor-state axiom* must be added at every time in the form:

$$F^{t+1} \Leftrightarrow ActionCausesF^t \lor (F^t \land \neg ActionCausesNotF^t) \, ,$$

where *ActionCausesF* is a disjunction of all the ground actions that have F in their add list, and *ActionCausesNotF* is a disjunction of all the ground actions that have F in their delete list;

- add *precondition axioms*: For each action A and each time, add the axiom

  $$A^t \Rightarrow \textsc{Pre}(A)^t,$$

  that is, if an action is taken at time t, then the preconditions must have been true.

- add *complete exclusion axiom* for every time: each pair of negated action must be inserted in a disjunction; this axiom ensure that only one action takes place at each time step, guaranteeing a totally ordered plan.

A conjunction of every result obtained from this translation represents the input for a SAT solver. If the formula is satisfiable, a list of actions can be extracted and sorted by their time index.
Observing the translation rules, it is easy to understand how the time variable heavily affects the problem. The maximum time index determines the maximum duration of the plan, in terms of number of actions, and how difficult is the problem, in terms of formula length. Naturally, this parameter must be able to be chosen by the user.

Chapter 2

Design

In this chapter the architectural design of the application is showed. *Model-View-Controller* is the architectural pattern that drives the design of the entire application.

2.1 Model

2.1.1 Problem Description

As described in the first chapter, the model must contains the main entities of the planning problem: states and actions. The UML class diagram in figure 2.1 represents the model and does not need a deep explanation. The *Context* class is the container in which we can find initial state, goal and actions available. Literal is the equivalent of a fluent or a ground term.

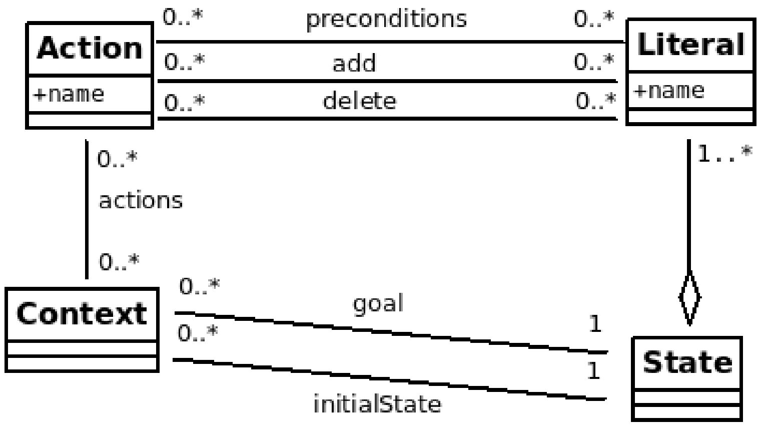


Figure 2.1: The model of the application.

2.1.2 Problem Resolution

After the design of the domain description, it is necessary to clear how to handle the planning process. To respect the principle of separation of concepts, it is important to separate the description of the problem, of which we dealt previously, from the particular resolution technique or technology. At this purpose, two interfaces has been created.
The *KBoundedPlan* interface defines a SAT-based plan description based on the entities showed in the previous section (Context, Action, State, Literal). The presence of the parameter X offers the possibility of coding the problem description in different ways. In fact, the *turnsIntoSatProblem* method can return a whatever specific representation of the domain, suitable for a specific solver that must be used. Thanks to this strategy, the general description of the problem is independent from a specific solver. The necessary translation can be done implementing the *turnsIntoSatProblem* method.
The *SATPlanSolver* represents a general interface for a solver based on a specific plan description.

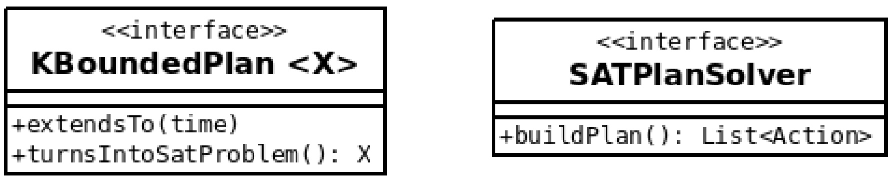


Figure 2.2: The part of the model dedicated to planning.

Chapter 3

Implementation

This chapter shows the main implementation choices made, following the design described in the past chapter. The entire code is available here as Gradle project hosted by a GitHub repository.

3.1 Technologies

The application has been written using Scala, a JVM-based programming language. Scala was preferred to other languages mainly for its complete and flexible collection framework.
The SAT solver has been taken from the *Tweety* framework. It is a comprehensive collection of Java libraries for logical aspects of artificial intelligence and knowledge representation. In particular, Tweety provides a module dedicated to the propositional logic that also contains a SAT solver. The available solver comes from another library called *Sat4J*. It is a full featured boolean reasoning library designed to bring state-of-the-art SAT technologies to the Java Virtual Machine. The Sat4J solver is based on the Minisat implementation.
All the third-part components used are open source.
The GUI has been built using JavaFX, the Java standard GUI library.

3.2 Integration with Sat4J

As mentioned in the previous chapter, the design of the application is completely independent of a specific SAT solver. Therefore, a specific *KBoundedPlan* has been instanced in order to create a description compliant to the library chosen.

In particular, the entire domain is translated into a string given to a solver instance. At this purpose, the builder pattern has been used. The *StateBuilder* and *ActionBuilder* are the two objects implemented in order to encode properly states and actions. The translation towards a SAT problem has been made following the rules showed previously in the first chapter.

3.3 Optimization

In planning problem turned into a SAT problem, the time is a crucial parameter. Without any kind of optimization, the translation has to be done considering the entire period of time between 0 and k. However, if the solution is less than k, in terms of actions, we ask the solver to solve a bigger problem than necessary. To avoid this situation, the SAT formula is sequentially increased with the next time expected, from 0 to k. If the solver find a consistent solution before the k^{th} iteration, the process is stopped and the solution is returned.
In addition, the n^{th} elements are added to the current and available formula incrementally, thanks to a caching mechanism.

3.4 Code Organization

The code is divided in several packages.
The *model* package contains the classes that are part of the model of the applications, as described in the second chapter. In addition, it contains a *plan* package in which we can find the interfaces for the SAT-based problem and the specific implementations for Sat4J. Finally, there is a package that contains all the exceptions created.
The *view* package is utterly dedicated to the user interface. It contains the JavaFX controllers and an object, called *LaunchScalaPlan*, that is the entry point for the application.
The *controller* package contains a single objects that allows the communications between the view and the model, just like the MVC pattern expects.
At the end, inside the *example* package there is a simple example useful to understand how the project works.

Chapter 4

Conclusions and Future Developments

In conclusion, a planning application called ScalaPlan has been developed. It solves the planning problem by translating it into a SAT problem and giving it to a proper solver. After a STRIPS-based description of the problem, the user obtains the sequence of actions that must be done to reach the goal, if it is exists. The application has been built using the Scala programming language and some additional libraries. It can be used from code but also a GUI-based version exists.
Since a low-level language such as C or C++ was not used, the application is certainly not one of the most performing. However, its being based on the JVM makes it portable and easy to use. It can provide a first and simple approach to the difficult problem of planning.
The introduction of variables inside the domain description seems to be a natural development for this project.

Bibliography

[1] D. L. Berre and A. Parrain. The sat4j library, release 2.2. *JSAT*, 7(2-3):59–6, 2010.

[2] R. E. Fikes and N. J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. In *Proceedings of the 2Nd International Joint Conference on Artificial Intelligence*, IJCAI'71, pages 608–620, San Francisco, CA, USA, 1971. Morgan Kaufmann Publishers Inc.

[3] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009.

[4] T. J. Schaefer. The complexity of satisfiability problems. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, STOC '78, pages 216–226, New York, NY, USA, 1978. ACM.

[5] M. Thimm. Tweety - a comprehensive collection of java libraries for logical aspects of artificial intelligence and knowledge representation. In *Proceedings of the 14th International Conference on Principles of Knowledge Representation and Reasoning (KR'14)*, July 2014.