Towards modularity for planning and robot programs verification

Author: Lilian Besson * **Tutors**: Jules Villard and Peter O'Hearn

Team: PPLV (Programming Principles, Logic and Verification Group)

University: University College London

June - August 2013

Synthesis

Global context

This document is designed to be the report for the internship I did at the end of the scholar year 2012–2013. As I intent to graduate this year a double degree in *Mathematics* and *Computer Science*, this internship was a short one (10 weeks), and counted for the two degrees (*Maths* "magistère" of `ENS Cachan`; and 1^{st}-year of the "MPRI" for *Computer Science*).

Between 03 June and 09 August 2013, I had the chance to work within the PPLV team, at `UCL` in London, UK.
I was tutored by Jules Villard and Peter O'Hearn (currently head of the team).

Scientific context

During my internship, I was mainly concerned with the links between `AI` and program verification, focusing on the *frame*, and *ramification*, problems, and the need for *modularity*. These problems come from both the domain of *deterministic* planning in `AI` (which is to find a trajectory to satisfy some goal in a dynamic world and the domain of program verification (which is to verify programs by finding and proving specifications), mainly formal verification methods based on Hoare logic or the more recent separation logic.

To be short, `AI` planning and program verification define the **frame problem** as *avoiding specifying the non-effects of actions* and the **ramification problem** as *understanding the indirect effects of actions*. And *modularity* is often translated as *procedural abstraction*: when a program has been proved, it can be used as a primitive element (and as if its specification was an axiom) anywhere, *without reusing the program's body*. Modularity can also be viewed in the programmer point-of-view: one would like to be able to write macros, functions, or libraries and reuse them, without having to know *how* exactly they have been written.

*Please, do not hesitate to contact me by *email* (at Lilian.BESSON[at]ens-cachan[dot]fr) for any question, any comment or to signal any problem about this internship report.

These two domains are both mature, and active since approximately 40 years. In particular, during the last 10 years, the work of Peter O'Hearn, David Pym and John C. Reynolds defines separation logic – a proof system based on Hoare logic; and from 1991 the work of Raymond Reiter and Fangzhen Lin defines the situation calculus. This formalism was used by the IPC (International Planning Competition), for a few years (2002 to 2008) which led to the axiomatization of the PDDL norm to describe planning domains and problems. But no links between those two domains have been really exploited with success yet.

Questions initially asked

One would like to know if recent verification techniques can help with AI problems, and conversely. At the least, one would like to know if the solutions proposed in one community can, in principle, solve the problems of the other. Verification is currently one of the most active research field in Computer Science; and planning is also an active research domain in AI, with in particular a couple of major *international competitions* focused on this very field. So, trying to prevent researchers of one field to reinvent the solution already used on the other field would help the two domains, may highlight some hidden links between them, or may even improve performances of concrete tools in AI or in verification.

But this is not an entirely new question, because by now ideas from planning have been used to boost the tractability of various program analysis tasks (as in the tool SpaceInvader). However, *basic* ideas from AI have been borrowed by verification research teams, with success, but more *technical* ideas have not been used, and it seems that there has been no efficient flow of ideas from verification to AI yet.

Proposed contribution

We have studied *state-of-the-art techniques* in both domains: PDDL, STRIPS and the situation calculus for AI planning; and Hoare logic and separation logic for program verification. For a more concrete approach, we have used two recent tools from AI planning domain: pyperplan to find a trajectory solving a fixed goal, and validate to prove his correctness. As links between AI and verification, the GOLOG language and its proof system HG (proposed by Liu in 2002) have also been studied.

First, to become more comfortable with planning and predicate calculi, we developed a few techniques, to increase the expressiveness of the language accepted by pyperplan (a small subset of PDDL). Then, we have been focused on the frame and ramification problems, but mainly on compositionality. For this, we have shown that the STRIPS formalism is weak against the frame problem, even if it relies on a frame hypothesis to implicitly solve the frame problem for primitive actions, although separation logic prefers a more syntactical approach. We also discovered this weakness against the frame problem in Liu's proof system for GOLOG programs, by pointing out a serious lack of modularity when dealing with non-primitive actions – *i.e.* user-defined procedures. Therefore, we tried to propose directions of research to improve modularity of this proof system HG, and thus for the GOLOG language.

Another weakness of the situation calculus and STRIPS formalism is a *completeness hypothesis*, which requires a perfect knowledge and a complete description of the studied domain. Sometimes, the difficulty is hidden behind this hypothesis, so the user may have to "work" a lot

to produce a suitable axiomatization, or to update an existing one. This is seen as a different kind of modularity. As a possible future work, we proposed the notion of *action refinements*, but we did not have time to develop it.

In the last days of the internship, we started to consider a way to model STRIPS in the separation logic, or to add a syntactical frame rule to STRIPS, stronger that the first frame hypothesis, used to describe separated items and solving planning problems more efficiently.

Arguments in favor of its validity

We have worked on many examples, showing weaknesses of a naive use of the PDDL formalism against large worlds, ramification, but most of all updates of an axiomatization. These weaknesses are *ipso facto* present in the solver pyperplan.

One of our contribution consists of some formal techniques to improve the method used to describe a dynamic world. They could result as a patch for pyperplan, to improve its performance on large words, and ramification; and to maybe improve its performance while updating an axiomatization (with refinements) or while unifying disjoint worlds (with the interpretation in separation logic). We have also shown the correctness of these techniques, by giving the intuition for acceptable bounds on both temporal and spatial complexities, to legitimate the use of these improvements for concrete examples (like for instance the freecell classic cardgame).

Sum-up and future

PDDL is very expressive, and thus covers many kind of domains. However, the pyperplan solver is quite limited, and supports only a small number of PDDL extensions (typed and positive-STRIPS only). Our approach tries to solve some of the limitations of this tool, and also some intrinsic weaknesses of the formalism. The techniques developped could have been implemented in pyperplan, but we did not, by lack of time. On the other hand, the lack of generality of this tool does not imply a lack of generality of our contribution, because the techniques improving the formalism are not limited the be applied with pyperplan, in particular the boolean reification algorithm.

With more time, we would have continued to work on modularity for GOLOG proof system and for updating a STRIPS-defined world; with the ideal goal of bringing a *good modularity* to both of those systems.

One of the next question we could ask is to know if the AI planning domain, translated in the separation logic point-of-view, is indeed efficient against the frame problem, and still usable in pratice. By now, the STRIPS formalism requires an implicit frame hypothesis while defining actions by a precondition and a postcondition, but we don't have any argument to justify that using partial models in separation logic and a formal separation conjunction $*$ is a good idea to scale while describing large separated domains, or to improve compositionality for the HG proof system; and this could be another possible direction of research.

1 Introduction

1.1 Scientific context and theme

One of the difficulties of this internship was to become familiar with two domains, relatively different from each other. Historically, Hoare logic is probably the most well-known formal method to ensure properties of programs, and has been first introduced by Tony Hoare in 1969 as one of the first **verification method** ([Hoa69]).

Moreover, John McCarthy was one of the pioneer in the domain of **artificial intelligence** (henceforth abbreviated by `AI`), when he started to use the predicate calculus to axiomatize changing worlds, in the late 60s ([MH68, part 3,4]). Predicate calculi have been studied from the 70s, in particular by Raymond Reiter and more recently by Fangzhen Lin, who continued to use and develop the *situation calculus* ([LR97, Rei01, Lin08]).

As a very interesting link between these two fields, Hector Levesque *et al.* have developed `GOLOG`, a language designed to write robot programs ([LRL$^+$97]), and, in 2002, Yongmei Liu endowed it with a proof system à-la Hoare logic ([Liu02]). Finally, the `STRIPS` formalism and its norm `PDDL`[1] – both inspired by the situation calculus – have been developed to concretely work on planning, and are the theoretical bases of tools like `blackbox` or `pyperplan`.

1.2 Issues and goals

One of the main question in `AI` is path finding, also called *planning*. In *graph theory*, **path finding** is a refinement of the question of reachability: in a graph $\mathcal{G} = (E, V)$, given two vertices $u, v \in E$, one would like to know if there is a path between u and v – and possibly to have one witness – for the binary relation "is-connected-to" $\rightsquigarrow$ (*i.e.* a sequence $(u_i)_{0 \leqslant i \leqslant n}$ with $u_0 = u, u_n = v$ and $\forall i.\, u_i \rightsquigarrow u_{i+1}$). This analogy is pursued in the appendix, section A.

Similarly, in `AI`, one can describe *states* of a dynamic world instead of vertices, and *transformations* on the world instead of edges. For example, we quickly present here the suitcase planning problem: from an initial state, u, where there is a closed suitcase s_1, a possible goal state, v, can be to open the suitcase. Then the *reachability* question for this small system is to know if it is possible to execute a *sequence* of **actions** from u to v. In this toy example, we need to open the suitcase, and this could be written with something like $Open(s_1)$. Another question is to prove the validity of such paths, *i.e.* to be sure that the actions can be effectively performed one after the other.

On the other hand, in software verification, the "path" is already there: it is the program itself (seen as a *sequence* of **instructions**); and one goal can be to find a proof of some program properties (written as a specification), or to mathematically validate a proof, or even to automatically generate a program specification.

Among all the characteristics a formalism could have, conciseness, clarity, and simplicity are the most appreciated in both verification and `AI`: the *simpler* the formulas are, the *easier* they can be understood and used by a human. But, we also want to be as expressive as possible, so there is always a balance between clarity and expressiveness. Therefore one of the goal of `AI`

[1] `STRIPS` means `STanford Research Institute Problem Solver`, and `PDDL` means `Planning Domain Definition Language`.

and verification is to design formalisms not only efficient and expressive but also concise and easy-to-work with.

1.3 Outline

In this report, we try to present some of the more interesting questions we had worked on.

First, we expose some simple examples to quickly show what kind of things we want to model. As a first changing world, we introduce a robot operating on contruction blocks on a floor, then to control this robot we write a first program, and we axiomatize its behavior by writing a formal specification. We shortly introduce the reader to two major problems: modularity and non-linearity, before presenting in a more rigorous way the different formalisms. The situation calculus is used to axiomatize dynamic worlds, the `GOLOG` language is used to program a robot operating of such worlds, and the `HG` proof system is designed to write and prove specifications of such programs. Then we show a serious weakness of this proof system, by writing two other programs on the block world, both using the first one. One of our contribution is to point out a lack of modularity for `HG`, because it does not handle user-defined procedures as well as primitive actions. We discuss the possible reasons of this weakness, and we also try to point out what could be considered as a solution.

In a second part of the report, we are focused on a more concrete approach to the planification problem, by exposing the `STRIPS` formalism – a subset of the situation calculus – and two tools used concreteley (`pyperplan` and `validate`) and some weaknesses of these elements. We also expose the results of some experiments designed to evaluate `pyperplan`, and then we develop a couple of formal and automatic techniques to improve the expressiveness of the `STRIPS` formalism.

In the last part, we will make a short sum-up, followed by a perspective and some possible future direction of research. An appendix is included, to present a partial bibliography, and two concrete axiomatization written in `STRIPS` according to the state-of-the-art norm called `PDDL`.

2 An informal presentation of what we want to model

We quickly introduce here what we want to be able to model with an example of a dynamic world, then we use it to write our first robot programs, and we present how to formally specify it by writing a specification.

2.1 The block world: a canonical example in planning

We would now like to present a simple example of what kind of "worlds" the `AI` domain is interested on. Let us imagine a clear floor, containing some construction blocks, and an ideal robot, *operating* on these blocks. The robot has two **actions**: it can move a block `A` onto `B` (this is written `Move(A,B)`), or can move `A` from `B` to the floor, with `PutFloor(A)`, and therefore it can build towers of blocks. We then use **predicates** to describe the relationships between blocks: the block `A` can be clear (*i.e.* `A` do not have anything on it) and this is written with a predicate (also called **relation fluent**) like `Clear(A)`. We also use `On(A,B)` to say that the block `A` is on `B`; and `OnFloor(A)` says that `A` is on the floor. Finally, we describe the world with a **state**, and for example one initial state can be 3 different blocks (`A, B, C`), each of them being clear and on the floor. This is written as a formula in a predicate logic:

$$\Gamma_0 \stackrel{\text{def}}{=} \big(\texttt{Clear(A)} \wedge \texttt{OnFloor(A)}\big) \wedge \big(\texttt{Clear(B)} \wedge \texttt{OnFloor(B)}\big) \wedge \big(\texttt{Clear(C)} \wedge \texttt{OnFloor(C)}\big) \quad (2.1)$$

And this can be illustrated as shown in this figure:

Figure 1: Initial state (S_0, Γ_0). (`A,B,C` are different from each other)

An achievable goal Now, we can ask the robot to build a tower of three blocks (`A` on `B`, `B` on `C`), *i.e.* we write $\Gamma_{goal} \stackrel{\text{def}}{=} \texttt{On(A, B)} \wedge \texttt{On(B, C)}$ as a **goal state**. We start from an empty **history** of executed actions (also called a **situation**[2]): $S_0 \equiv [\,]$. We say an action is **executable** if it can be performed, for example, `Move(A,B)` requires `A` and `B` to be different and both clear (*i.e.* $(\texttt{A} \neq \texttt{B}) \wedge \texttt{Clear(A)} \wedge \texttt{Clear(B)}$). And one of its consequences is to effectively move `A` on top of `B`, so **performing** the action `Move(A,B)` will allow to deduce `On(A,B)`.

Therefore, a **valid** solution to the *path finding* problem for this domain could be to put `B` on `C` (going from S_0 to S_1, written $S_0 \rightsquigarrow S_1$), with `Move(B,C)`, and then `A` on `B` ($S_1 \rightsquigarrow S_2$). Then the final situation S_2 is the history expressing what we have done to arrive in Γ_{goal}; and this is written as follow:

$$S_1 \stackrel{\text{def}}{=} Do(\texttt{Move(B, C)}, S_0), \quad (2.2)$$

$$S_2 \stackrel{\text{def}}{=} Do(\texttt{Move(A, B)}, S_1). \quad (2.3)$$

[2]This is formally defined after, see definition 3.1

Figure 2: A goal state Γ_{goal}, achieved by S_2.

An unachievable goal Another goal Γ'_{goal} could be to have `A` on `B`, `B` on `C` (like before), but to require `C` on `A` additionally. This seems to be impossible, or at least strange, and contradict the intuition we have about these blocks; and we will see that a good formalization of the block world allows to prove the **non-reachability** of this strange goal. We cannot embed a picture representing this second goal, because of this "irrational" relationship.

We will explain how to **axiomatize** this domain and the two planning problems in section 3.1; by doing so, we will softly introduce the situation calculus formalism.

Remark 1 (Only deterministic worlds)**.** *The formalism we will present is also able to represent stochastic actions and states, but we preferred to stay focused on the simplest sub-domain: only deterministic worlds. This restriction is also done in some major works on planning, like [Lin08].*

However, an introduction to the stochastic point-of-view can be found in [Rei01, part 12].

2.2 Robot programs operating on a dynamic world

This block world is a good example of a simple but interesting **dynamic world**, and one could ask if it possible to write procedures operating on such a world. The robot we have presented before can move a block onto another, or on the floor but it does not have any more sophisticated actions. Now, we will quickly use the `GOLOG` language ([LRL+97]) to write more complicated procedures for this robot, *i.e.* to write **robot programs**.

Our first robot program: `MoveOr2` As his name suggests it, the procedure `MoveOr2` is designed to **move** a block x on top of another, non-deterministically chosen (`ND`) between two other blocks y, z:

$$\texttt{Proc MoveOr2}(x, y, z) \texttt{ : Move}(x, y) \texttt{ | Move}(x, z) \texttt{ EndProc;} \tag{2.4}$$

From here on, we use the shortcuts $\delta_1 \stackrel{\text{def}}{=} \texttt{Move}(x, y)$ and $\delta_2 \stackrel{\text{def}}{=} \texttt{Move}(x, z)$ to denote the left and right parts. Hence, $\delta_1 \,|\, \delta_2$ means a *non-deterministic* choice between δ_1 and δ_2. So, we want to prove that `MoveOr2` effectively moves x to y **or** to z. We will write a partial specification for `MoveOr2` in the next subsection (*c.f.* formula (2.5)) and prove it after (*c.f.* proof 1).

High-level robot programming This language is like a bridge between verification and `AI`: it allows one to write programs talking about a user-defined dynamic world, and then to prove programs specifications, expressed in an "assertion language". This kind of robot programming is often called high-level robot programming, because we are not interested on how exactly the primitive actions are executed, we are just interested on how they interact with each other in a

robot procedure. Hence, we precise that our "robots" are only a theoretical model, we were not interested in any kind of concrete machineries.

2.3 How to write and prove program properties

Hoare logic ([Hoa69]) is a well-known formalism to express and prove specifications for programs, with formulas written in first-order logic. We assume that the reader is at least familiar with usual first-order logic, thus it will not be re-introduced precisely in this report. We introduce here an example of partial specification, for the procedure `MoveOr2` previously defined.

`MoveOr2` is a procedure, but as primitive actions it requires some *conditions* to be performed. First, we need $(x \neq y)$ and $(x \neq z)$ to be sure that x can indeed be moved onto a *different* block y or z. To be executable, the first part δ_1 requires $\texttt{Clear}(x) \wedge x \neq y \wedge \texttt{Clear}(y)$, when δ_2 requires $\texttt{Clear}(x) \wedge x \neq z \wedge \texttt{Clear}(z)$. Hence, after simplification, we would like to have the following partial[3] specification:

Lemma 1 (Specification for `MoveOr2`)**.** *We prove it after, see proof 1.*

$$\{\underbrace{\texttt{Clear}(x) \wedge (x \neq y) \wedge \texttt{Clear}(y) \wedge (x \neq z) \wedge \texttt{Clear}(z)}_{\stackrel{\text{def}}{=} Q_{Or2}}\} \quad \texttt{MoveOr2}(x, y, z) \quad \{\underbrace{\texttt{On}(x, y) \vee \texttt{On}(x, z)}_{\stackrel{\text{def}}{=} R_{Or2}}\} \tag{2.5}$$

What does procedural abstraction mean? Remark that we could have switch δ_1 and δ_2 in the body of `MoveOr2`, without changing its effect. So, if we write a second implementation of `MoveOr2`, with $\delta_2 | \delta_1$ instead, then we could give it the same specification. And thus, if we have a good **procedural abstraction**, we should be able to use the new implementation instead of the first one, reprove its specification, but without having to reprove anything else.

2.4 Two main problems

For programming in general, modularity is needed, because it is unthinkable to force a programmer to in-line all the procedure he want to define. Thus, **programming compositionality** appeared with the first programming languages, and is still present in almost every languages (even LaTeX!). For instance, we defined `MoveOr2`, and then we will use it to construct bigger programs, without having to expand it or rewrite it wherever it will be used.

Modularity for verification To formally prove specifications for those kind of procedures, the key idea is to have a certain **modularity** in the underlying logics: the specifications are written for each primitive elements, and each transition from one element to the next is proved with axioms or inference rules. The proof system is a set of formal rules (see next section 3.3 for examples of such inference rules). Compositionality means that when a sub-procedure has been proved, one could use his specification everywhere it is used, *without having to reuse the procedure's body* nor having to reprove it somewhere else. In fact, **procedural compositionality** is one of the more needed property for a proof system or for a practical analysis tool. For example, the tool `Abductor` has *successfully* verified subsets of very huge projects like `Apache` or the `Linux` kernel, because it is compositional (complete benchmarks for those experiments are in [CDOY11]).

[3] Only partial because we do not specify its effect on x's initial position (either $\texttt{OnFloor}(x)$ or $\texttt{On}(x, z)$ is transformed from true to false).

Non-linearity? But we also need to be able to change an axiomatization a little bit, without having to rewrite everything, and to add one element without having to reconsider all what have already be done. This is related to the notion of linearity in logics, and in fact a formalism designed to axiomatize a dynamic world is often **non-linear**: adding knowledge to the system can reduce what we know, *i.e.* what we are able to prove in the system. One example of such non-linear behavior can be given with the block world again. If we want to add a new predicate `HasMoved` to know if a block has been moved from the initial state (Γ_0, S_0), we can add $\texttt{HasMoved}(x)$ as an additional effect of $\texttt{Move}(x, y)$ and of $\texttt{PutFloor}(x)$. Then, this additional information about the world can weaken some specifications already proved.

3 Three formalisms to prove robot programs

3.1 A quick presentation of the situation calculus

The situation calculus is a many-sorted second-order language designed to represent dynamic worlds. It uses 4 disjoint sorts: `relational fluents`, `objects`, `situations`, and `actions` which have already been informally introduced in section 2.1. The formalism we will present in this section is used to axiomatize deterministic dynamic world, and the result of such an axiomatization is a called **basic action theory**, denoted by a symbol $\mathcal{D}$.

For example, in the block world, we recall that **action symbols** are `Move` and `PutFloor`, of arity 2 and 1; **relational fluents** are `OnFloor` of arity 1 or `On` of arity 2. We will use A to design an action symbol, and F to design a relational fluent. They will be written as[4] $A(x_1, \ldots, x_n)$ if A has an arity n, and $F(y_m, \ldots, y_m)$ if F has an arity m. And, x_i and y_j are used to denote objects, like `A` or `C` in the block world. We often use the shortcuts $\overrightarrow{x} \stackrel{\text{def}}{=} x_1, \ldots, x_n$ or $\overrightarrow{x_i} \stackrel{\text{def}}{=} (x_i)_{1 \leqslant i \leqslant n}$. We precise that arities can be equal to zero, *i.e.* $n, m \geqslant 0$.

And finally, **situations** s are *recursively* defined as follow, with S_0 the empty situation:

$$s ::= S_0 \mid Do(A(\overrightarrow{x}), s). \tag{3.1}$$

We need to express the fact that an action can be **performed**, so let $Poss(a, s)$ be a binary predicate meaning that action a can be applied in situation s (*i.e.* is *executable*), with a an *instantiated* action, of the form $A(\overrightarrow{y_j})$, and s a situation, We are only interested in *executable situations*, *i.e.* histories in which it is possible to perform the actions one after the other:

$$s ::= S_0 \mid Do(A(\overrightarrow{x}), s), \qquad \textbf{if } Poss(A(\overrightarrow{x}), s). \tag{3.2}$$

The situation calculus allow to consider first-order formulas, of the following form (3.3), where all the classical logical connectors are interpreted *as usual* ($\wedge$, $\vee$, $=$, $\neq$, $\neg$, $\exists$, $\forall$, $\Rightarrow$) and with $(\phi_1 \equiv \phi_2) \stackrel{\text{def}}{=} (\phi_1 \Rightarrow \phi_2 \wedge \phi_2 \Rightarrow \phi_1)$:

$$\phi ::= (\phi) \mid F(\overrightarrow{y}) \mid \overrightarrow{x} = \overrightarrow{y} \mid A(\overrightarrow{x}) = A'(\overrightarrow{x}) \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \Rightarrow \phi \mid \exists x.\, \phi(x) \mid \forall x.\, \phi(x) \tag{3.3}$$

So for the block world we can write axioms like $\forall x.\ (\texttt{Clear}(x) \vee \exists y.\, \texttt{On}(y, x))$, to say that either a block x is clear, either there is another block y on top of x. In the first initial world we presented above, Γ_0 only contained zeroth-order formulas (like `Clear(A)`). But the domain can also contain first-order formulas, to add some **domain constraints**. For example, one would like to have $\texttt{On}(x, y) \Rightarrow \neg\texttt{On}(y, x)$ and $\exists y.\, \texttt{On}(y, x) \Rightarrow \neg\texttt{Clear}(x)$ as domain constraints for the block world. We only ask Γ_0 to be **coherent**, *i.e.* we do not want to have $\Gamma_0 \vDash$ **false** in the underlying logic. We would like to precise that the variables not explicitly existentially quantified are implicitly universally quantified, so we do not write every $\forall x, y$ in each formulas, if it is not ambiguous.

[4] A convention is to use names construct as a short version of the concatenation of the natural name for the action or the predicate, and use capital letter to denote a new world ("put on the floor" becomes `PutFloor` for example).

Remark 2 (Extra predicates as shortcuts)**.** *For the block world, we used 3 different relational fluents, but they are* not *independents because we introduced some domain constraints. In fact, it is often possible to express some predicates with a smaller subset of predicates (in this case, just* `On`*). For example, with first-order formulas, we can write* $\texttt{Clear}(x) \stackrel{\text{def}}{=} \neg\big(\exists y.\, \texttt{On}(y,x)\big)$*, and* $\texttt{OnFloor}(x) \stackrel{\text{def}}{=} \neg\big(\exists y.\, \texttt{On}(x,y)\big)$*.*

There is here two different point-of-views: if we use all the predicates, such equivalence formulas can be proved (for all situations), by using some domain constraints and the classical proof system for first-order formulas. But if we define some predicates as shortcuts, then some domain constraints become provable, like for instance this one: $\exists y.\, \texttt{On}(y,x) \Rightarrow \neg\texttt{Clear}(x)$ *(presented in the previous paragraph)*

A relational fluent $F(\overrightarrow{x})$ can be written with a situation s as its last argument $\big(F(\overrightarrow{x}, s)\big)$ to consider the fact $F(\overrightarrow{x})$ *in the situation* s. In our block world, we had $\texttt{Clear}(\texttt{C}, S_0)$ but then we applied the first action to go from S_0 to S_1, and hence we no longer had $\texttt{Clear}(\texttt{C}, S_1)$.

We say that ϕ is a **pseudo-fluent formula** (*pff*) if all the fluents in ϕ have no situation term. In this case, we write $\phi[s]$ the new formula obtained from ϕ by adding s as the last argument in every fluent. For example, $\phi \stackrel{\text{def}}{=} \texttt{Clear}(x) \wedge \neg\texttt{OnFloor}(x)$ is a *pff* and so $\phi[s] = \texttt{Clear}(x,s) \wedge \neg\texttt{OnFloor}(x,s)$.

Now, we need to describe **when** an action A can be performed in a situation s. This will be written as one axiom for each action A, of the form $Poss(A(\overrightarrow{y}), s) \equiv \Pi_A(\overrightarrow{y})[s]$, where Π_A is a *pff*. For instance, for `Move`, $Poss(\texttt{Move}(x,y), s) \stackrel{\text{def}}{=} \texttt{Clear}(x,s) \wedge (x \neq y) \wedge \texttt{Clear}(y,s)$, and for `PutFloor` $Poss(\texttt{PutFloor}(x), s) \stackrel{\text{def}}{=} \exists y \neq x.\, \big(\texttt{Clear}(x,s) \wedge \texttt{On}(x,y,s)\big)$. Those axioms are called **action precondition axioms**, and we have to give one for every action a. Moreover, we require that those axioms describe *completely* the minimal condition under which an action can be performed. This is the first **completeness assumption**.

Now that we can describe **when an action** can be executed, we need something to describe the **effect**(s) of executing an action.

Remark 3 (Duality action-predicate)**.** *Here appears a major* duality *between relational fluents and actions because there is two possible approaches now. One could choose to write:*

(EF) *one axiom* $Post(a)$ *for* each action a*, with a list of the fluents changed by this action (*`STRIPS` *choice, detailed after in* (5.4) *and* (5.5)*), or*

Post(a) *one axiom* Φ_F *for* each predicate F*, with a "case-by-case" analysis of the action (situation calculus choice).*

In the situation calculus, we use **successor state axioms**, *i.e.* **one axiom** of the form $F(\overrightarrow{y}, Do(a,s)) \equiv \Phi_F(\overrightarrow{y})[s]$, – for each fluent F – to characterize *exactly*[5] how the fact $F(\overrightarrow{y})$ can be true after performing any action a. Here we can use case-by-case analyzes of the form $\big(a = A(\overrightarrow{x}) \wedge \ldots\big)$. Basically, $\Phi_F(\overrightarrow{y})$ has two parts, one saying that $F(\overrightarrow{y})$ is true after performing a if a does not modify it, and $F(\overrightarrow{y})$ was true before (**non-effect**); and another part saying that $F(\overrightarrow{y})$ is true after applying a if a makes it become true (**effect**).

For example, in our block world, we will axiomatize the effects of the two actions `Move` and `PutFloor` with one successor state axiom for every predicates, *i.e.* by writing one Φ_F for F

[5]This "*exactly*" is the second part of the **completeness assumption** made by the situation calculus.

being `Clear`, `On`, and `OnFloor`, as follow:

$$\begin{aligned}
\texttt{On}(x, y, Do(a, s)) &\equiv a = \texttt{Move}(x, y) \vee \big(\texttt{On}(x, y, s) \wedge a \neq \texttt{PutFloor}(x) \wedge \neg(\exists z \neq x, y.\, a = \texttt{Move}(x, z))\big);\\
\texttt{OnFloor}(x, Do(a, s)) &\equiv (a = \texttt{PutFloor}(x)) \vee \big(\texttt{OnFloor}(x, s) \wedge \neg(\exists y \neq x.\, a = \texttt{Move}(x, y))\big).\\
\texttt{Clear}(x, Do(a, s)) &\equiv \exists y \neq x.\, \Big(\big(\,(\exists z \neq x.\, a = \texttt{Move}(y, z)) \vee a = \texttt{PutFloor}(y)\big) \wedge \texttt{On}(y, x, s)\Big)\\
&\qquad \vee \big(\texttt{Clear}(x, s) \wedge \neg\exists y.\, a = \texttt{Move}(y, x)\big);
\end{aligned}$$

Finally, we recall that the **goal state** (denoted Γ_{goal}) is also a set of formulas, where we can also use quantifiers.
For instance, a different goal for the block world could be to simply require to have *at least* a tower of three blocks, and this can be written as $\Gamma'_{goal} \stackrel{\text{def}}{=} \exists x \neq y \neq z.\, \texttt{On}(x, y) \wedge \texttt{On}(y, z)$.

As a last remark, we recall that free variables are also allowed in the description of actions. That is why we can reduce the signature of actions to their minimal sizes: and for example $\texttt{PutFloor}(x)$ just needed one argument (the block x being moved) and the block y on which x were is introduced with a $\exists y$ in $\Pi_{\texttt{PutFloor}}$.

Remark 4 (Unique name axioms)**.** *A classical assumption made in the situation calculus is the* unique names axioms*, for "everything". This means,* $a(\overrightarrow{x}) = a'(\overrightarrow{y}) \Rightarrow (a = a' \wedge \overrightarrow{x} = \overrightarrow{y})$*, for every actions* a, a'*, and their arguments* $\overrightarrow{x}, \overrightarrow{y}$ *and* $F(\overrightarrow{x}) = F'(\overrightarrow{y}) \Rightarrow (F = F' \wedge \overrightarrow{x} = \overrightarrow{y})$*, for every fluents* F, F'*, and their arguments* $\overrightarrow{x}, \overrightarrow{y}$*. Finally, we have 2 axioms to complete the definition of situations:* $\forall A, A', s, s'.\, Do(A, s) = Do(A', s') \Rightarrow \big(A = A' \wedge s = s'\big)$*, and* $\forall A, s.\, Do(A, s) \neq S_0$.

3.2 A language based on user-defined primitives

We can quickly define the `GOLOG` language as an `Algol`-like language, where the primitives are *user-defined* actions, and where the programs operates on a user-defined dynamic world completely axiomatized in the formalism of the situation calculus.

In a classical imperative programming language, the underlying *dynamic world* is usually the memory, so the objects being manipulated are variables and values (integer or floats) and the smallest elements of programs are **primitives**, like writing a value v on the memory case x (often written $x := v$), or reading the value of x ($!x$); or even arithmetical operations (like $v_1 + v_2$). And then, programs are built inductively, for example by composing sequentially: $\delta_1 ; \delta_2$ (to do δ_1 **and then** δ_2) or by a non-deterministic[6] choice: $\delta_1 | \delta_2$ (to do δ_1 **or** δ_2).
Other **constructions** exist, as the non-deterministic iteration δ^*, to iterate δ a unspecified number of time, or the more unusual $(\pi x)\delta(x)$ designed to non-deterministically choose an argument x for the body δ.

The `GOLOG` language ([LRL+97]) uses the same kind of rules to inductively build programs, but it differs from classical languages by using *user-defined* **primitives**. This simple idea states that, for instance, moving a block onto another (`Move`) is nothing else but a primitive of the language. And that is why we used the composition rule and two primitives $\texttt{Move}(x, y)$ and $\texttt{Move}(x, z)$ to write $\texttt{MoveOr2}(x, y, z)$, in section 2.2.

[6] We use 3 different constructions based on non-determinism, but we recall here that we *only* consider deterministic worlds.
The non-determinism introduced here is just for programs, *never* for actions or for states.

The grammar for GOLOG programs is presented below:

$$\begin{aligned}\delta \quad ::= \quad & (\delta) \mid A(\overrightarrow{v}) \mid \phi? \mid \delta;\delta \mid \delta \,|\, \delta \mid (\pi x)\delta(x) \mid \exists x.\ \delta(x) \mid \forall x.\ \delta(x) \mid \delta^* \mid \\ & P(\overrightarrow{v}) \mid \text{Proc } P_1(\overrightarrow{v_1})\ \delta_1\ \text{endProc}; \ldots ; \text{proc } P_m(\overrightarrow{v_m})\ \delta_m\ \text{endProc}\ \ \delta \end{aligned} \tag{3.4}$$

where ϕ is a *pff*, A is an action, $P, P_1, \ldots, P_m$ are procedures names all different from each other, with $m \geqslant 1$, and where $\delta(x)$ means that x appears free in δ.

The last construction, abbreviated as $\{Env;\ \delta\}$, provides **programming modularity**. We can use previously defined procedures P_i as if it was a primitive element in a new GOLOG program δ: these procedures $(P_i)_i$ just have to be included in an environment: defined as $Env \stackrel{\text{def}}{=} \big(\text{Proc } P_i(\overrightarrow{v_i})\ \delta_i\ \text{endProc};\big)_{1 \leqslant i \leqslant m}$.

This language can be used to define more sophisticated robot procedures. We already wrote MoveOr2 as a first example but conditionals and loops can be defined as simple abbreviations:

$$\text{if } \phi \text{ then } \delta_1 \text{ else } \delta_2 \ \stackrel{\text{def}}{=}\ \big(\phi?;\delta_1\big) \,|\, \big((\neg\phi)?;\delta_2\big) \tag{3.5}$$

$$\text{while } \phi \text{ do } \delta \text{ done} \ \stackrel{\text{def}}{=}\ \big(\phi?;\delta\big)^*;\big(\neg\phi\big)? \tag{3.6}$$

Then, one could be interested in formally proving some properties about such robot programs. For instance we would like to prove that the little procedure MoveOr2 really works as it is meant to, by proving Lemma 1.

3.3 The HG proof system for robot programs

To prove such specifications, we will use a proof system, called HG, inspired from classical Hoare logic, for robot programs written in GOLOG. It is is due to Liu, so for more details see [Liu02] (H is for Hoare, G is for GOLOG). We will not present the underlying semantics for GOLOG programs, given by Liu to formally present the semantics of the inference rules, but simply the basic ideas of this proof system. As in classical Hoare logic, we will write **Liu-triples** of the form $\{P\}\ \delta\ \{Q\}$, when P, Q are pseudo-fluent formulas (*pff*s) and δ is a GOLOG program.

The different between Hoare-triples and Liu-triples is how we **interpret** them: usually, $\{P\}\ \delta\ \{Q\}$ means that if Q is true, δ can be executed and *if it terminates*, then the system will be in a new state with R being true. But in the HG proof system, we only consider Q in *executable situations* s: if $Q[s]$ is true, then δ can be performed to change the system from s to s'. And if δ terminates when starting from s, performing it implies $R[s']$.

As the GOLOG language, its proof system HG is **parametric** in a basic action theory. Liu used the symbol $\mathcal{D}$ to denote an basic action theory (*i.e.* a dynamic world), and $\text{HG}(\mathcal{D})$ to denote the proof system for *this* dynamic world. For example, the block world will have one proof system $\mathcal{D}_1$, and the suitcase example (presented in the introduction) defines $\mathcal{D}_2$. HG is based on inference rules and primitive axioms, like the following:

(EF) We can quickly define the **frame problem** as *avoiding specifying the non-effects of actions*. These **Effect and Frame Axioms** come from the work of Reiter about frame ([Rei01]), and have not to be written by hand (HG use them without having to generate them):

$$\{\Phi_F(\overrightarrow{x}, A(\overrightarrow{y}))\} \quad A(\overrightarrow{y}) \quad \{F(\overrightarrow{x})\}, \tag{3.7}$$

$$\{\neg\Phi_F(\overrightarrow{x}, A(\overrightarrow{y}))\} \quad A(\overrightarrow{y}) \quad \{\neg F(\overrightarrow{x})\}, \tag{3.8}$$

where A is an action symbol, F is a relational fluent with *successor state axiom* $F(\overrightarrow{x}, Do(a,s)) \equiv \Phi_F(\overrightarrow{x}, a)[s]$. As their names imply it, these axioms embed a first solution to the *frame problem*, **for primitive actions only**; because we will have axioms like $\{F\}\ A\ \{F\}$ (resp. $\{\neg F\}\ A\ \{\neg F\}$) if performing A does not change the truth value of F (resp. of $\neg F$).

(FF) Fluent-Free Axiom, where Q does not contain any predicates, *i.e.* is of the form $\overrightarrow{x} = \overrightarrow{y}$ or $\overrightarrow{z} \neq \overrightarrow{w}$:

$$\{Q\}\ \delta\ \{Q\}. \tag{3.9}$$

(TA) These **Test Action Axioms** are the natural specifications for a requirement of the form $\phi?$, with any R:

$$\{\phi \Rightarrow R\}\ \phi?\ \{R\}. \tag{3.10}$$

Inference rules We quote here all the inference rules except PAR and RC. We expose only the rules used for the specifications proved after (*c.f.* the proofs 1, 2 and 3). The first two rules are only syntactical, and it is important to notice the subtle difference between them: (Seq) requires to have a shared R as $Post(\delta_1)$ and $Prec(\delta_2)$, because the sequence $\delta_1;\delta_2$ does δ_1 *then* δ_2; while (NA) requires to have the *same* specification for the two parts δ_1 and δ_2, because it does δ_1 *or* δ_2 "simultaneously".

2: Sequence (Seq)
$$\frac{\{P\}\ \delta_1\ \{R\} \qquad \{R\}\ \delta_2\ \{Q\}}{\{P\}\ \delta_1;\delta_2\ \{Q\}}$$

3: Non-deterministic Action (NA)
$$\frac{\{P\}\ \delta_1\ \{Q\} \qquad \{P\}\ \delta_2\ \{Q\}}{\{P\}\ \delta_1|\delta_2\ \{Q\}}$$

Figure 3: Some structural rules for the GOLOG programming language.

Before introducing the next rules, we need the notion of **executable truth**: for a *pff* Q, we write $\Box Q$ as a formula true **iff** Q is true in all *executable* situations (*i.e.* $\Box Q \stackrel{\text{def}}{=} \forall s.\ (executable(s) \Rightarrow Q[s])$).
Therefore, we give an additional axiom **(Inv)**, **parametric** in $\mathcal{D}$ because it uses the knowledge contained in the theory as domain constraints or facts assumed true in the initial state (*i.e.* included in Γ_0):

(Inv) "Invariant Oracle Axiom": $\Box Q$, if the theory $\mathcal{D}$ is sufficient to prove $\Box Q$. For example, in $\mathtt{HG}(\mathcal{D}_1)$, we have $\Box(\mathtt{On}(x,y) \Rightarrow \neg\mathtt{On}(y,x))$ because it is a domain constraint.

4: Conjunction (Conj)
$$\frac{\{Q_1\}\ \delta\ \{R_1\} \qquad \{Q_2\}\ \delta\ \{R_2\}}{\{Q_1 \wedge Q_2\}\ \delta\ \{R_1 \wedge R_2\}}$$

5: Disjunction (Disj)
$$\frac{\{Q_1\}\ \delta\ \{R_1\} \qquad \{Q_2\}\ \delta\ \{R_2\}}{\{Q_1 \vee Q_2\}\ \delta\ \{R_1 \vee R_2\}}$$

6: Consequence (Cons)
$$\frac{\Box(Q \Rightarrow Q_1) \qquad \{Q_1\}\ \delta\ \{R_1\} \qquad \Box(R_1 \Rightarrow R)}{\{Q\}\ \delta\ \{R\}}$$

Figure 4: Classical logical rules. (Cons) uses a $\Box$ to consider $\Rightarrow$ only in executable situations.

The next rules ∃I and ∀I are the usual introduction of variables quantifiers (from Hoare logic):

7: Introduction of ∃ (∃I)
$$\frac{\{Q\}\ \delta(x)\ \{R\}}{\{\exists x.\, Q\}\ \delta(x)\ \{\exists x.\, R\}}$$

8: Introduction of ∀ (∀I)
$$\frac{\{Q\}\ \delta(x)\ \{R\}}{\{\forall x.\, Q\}\ \delta(x)\ \{\forall x.\, R\}}$$

Figure 5: $\exists x$ and $\forall x$ Introduction rules with x free in Q, R..

The next two rules (NAA) and (NI) are relatively straightforward, and they state that iterating or choosing an argument does not change the specification, where $\delta(x)$ denotes a GOLOG program with a free variable x.

9: Non-deterministic Action Argument (NAA)
$$\frac{\{Q\}\ \delta(x)\ \{R\}}{\{Q\}\ (\pi x)\delta(x)\ \{R\}}$$

10: Non-deterministic Iteration (NI)
$$\frac{\{Q\}\ \delta\ \{R\}}{\{Q\}\ \delta^*\ \{R\}}$$

Figure 6: Non-deterministic rules (with x free in Q, R.).

The last rule (IK) we want to present is there to add a certain modularity by allowing to reuse a specification already proven, only where it is encountered *exactly as proven*. So, the invocation rule is designed to **propagate** the environment Env in every sub-parts of a procedure.

Informally, it just states that when we have $\{Env; P(\overrightarrow{v})\}$, we can replace every procedure call $P_i(\overrightarrow{v_i})$ in $P(\overrightarrow{v})$ by its contextualized version $\{Env; P_i(\overrightarrow{v_i})\}$ This is denoted by $P(\overrightarrow{v})[d \leftarrow \{Env; d\}]$ with $[d \leftarrow d']$ to substitute d by d' in δ.

11: Invocation Rule(IK)
$$\frac{\{Q\}\ P(\overrightarrow{v})[P_i(\overrightarrow{v_i}) \leftarrow \{Env; P_i(\overrightarrow{v_i})\}]\ \{R\}}{\{Q\}\ \{Env;\ P(\overrightarrow{v})\}\ \{R\}}$$

Figure 7: Programming compositionality (without recursion).

Finally, we precise that GOLOG supports mutually recursive procedures, but we do not use recursion in our examples, and we think writing (PAR) or (RC) is confusing. Indeed, we are mainly concerned about *how modular this proof system can be*, and in the next section we prove its non-modularity in a counter-example.

4 Pointing out the absence of modularity for HG

Here we consider again the block world, and we use our first simple GOLOG program MoveOr2, to define two other programs: MoveOr3 and MoveThen. We will use Liu's proof system to prove the specification for MoveOr2 (Lemma 1), and when we prove the specification of MoveThen we will show that this proof system is not really modular, because we will have to reuse the body of MoveOr2 to prove MoveThen.

4.1 A procedure is not like a primitive action

We can use the intermediate procedure MoveOr2 to define two new procedures:

MoveOr3 is designed to move non-deterministically a block x onto another, chosen between y, z, **or** w:

$$\text{Proc } \texttt{MoveOr3}(x,y,z,w) \text{ : } \texttt{MoveOr2}(x,y,z) \text{ | } \texttt{Move}(x,w) \text{ EndProc;} \tag{4.1}$$

MoveThen is designed to move x on y or on z (non-deterministically), **and then** move it to w:

$$\text{Proc } \texttt{MoveThen}(x,y,z,w) \text{ : } \texttt{MoveOr2}(x,y,z) \text{ ; } \texttt{Move}(x,w) \text{ EndProc;} \tag{4.2}$$

We want to prove the following (partial) specifications:

Lemma 12. *Env, Env′ are the environments defining* MoveOr3 *and* MoveThen*, and the common precondition* $Q_{Or3} \stackrel{\text{def}}{=} Q_{Then} \stackrel{\text{def}}{=} Q \stackrel{\text{def}}{=} \texttt{Clear}(x) \wedge (x \neq y) \wedge \texttt{Clear}(y) \wedge (x \neq z) \wedge \texttt{Clear}(z) \wedge (x,y,z \neq w) \wedge \texttt{Clear}(w)$*:*

$$\{Q_{Or3}\} \quad Env; \texttt{MoveOr3}(x,y,z,w) \quad \{\overbrace{\texttt{On}(x,y) \vee \texttt{On}(x,z) \vee \texttt{On}(x,w)}^{\stackrel{\text{def}}{=} R_{Or3}}\} \tag{4.3}$$

$$\{Q_{Then}\} \quad Env'; \texttt{MoveThen}(x,y,z,w) \quad \{\underbrace{\texttt{On}(x,w)}_{\stackrel{\text{def}}{=} R_{Then}}\} \tag{4.4}$$

Let recall here that $\texttt{Move}(x,w)$ is a primitive action, but $\texttt{MoveOr2}(x,y,z)$ is a call to an intermediate procedure, and so they only differ by the fact that we have the Effect and Frame Axiom (EF) for $\texttt{Move}(x,w)$ (with $A = \texttt{MoveToBack}$), but not for $\texttt{MoveOr2}(x,y,z)$. We can therefore easily describe non-effects for $\texttt{Move}(x,w)$, but not for a non-primitive element such $\texttt{MoveOr2}(x,y,z)$.

4.2 Proving MoveThen to point out the lack of procedural abstraction for HG

We quickly prove here the partial specification for the GOLOG procedures MoveOr2, MoveOr3 and MoveThen, by using the HG proof system presented before (in section 3.3).

4.2.1 A partial specification for `MoveOr2`

Proof 1 (Proof of Lemma 1)**.** *We shorten this* proof tree *and the next ones by using* $\varphi(a,b) \stackrel{\text{def}}{=} \Phi_{Move}(a,b) = \texttt{Free}(a) \wedge a \neq b \wedge \texttt{Free}(b)$ *to denote the precondition for the primitive action* $\texttt{Move}(a,b)$.

We use (NA) *to separate* δ_1 *and* δ_2. *Then we use* (Cons) *to weaken the preconditions (*Q_{Or2} *becomes* $\varphi(x,y)$ *or* Q_{Or2} *becomes* $\varphi(x,z)$*) and strengthen the postcondition (*R_{Or2} *becomes* $\texttt{On}(x,y)$ *or* $\texttt{On}(x,z)$*). Finally, we use the Effect Axiom* (EF) *for* $F = \texttt{On}$ *and* $A = \texttt{Move}$.

$$\dfrac{\dfrac{\dfrac{}{\{\varphi(x,y)\}\ \mathit{Move}(x,y)\ \{\mathit{On}(x,y)\}}\,\text{EF}}{\{Q_{Or2}\}\ \mathit{Move}(x,y)\ \{R_{Or2}\}}\,\text{Cons} \qquad \dfrac{\dfrac{}{\{\varphi(x,z)\}\ \mathit{Move}(x,z)\ \{\mathit{On}(x,z)\}}\,\text{EF}}{\{Q_{Or2}\}\ \mathit{Move}(x,z)\ \{R_{Or2}\}}\,\text{Cons}}{\{Q_{Or2}\}\ \mathit{Move}(x,y) \,|\, \mathit{Move}(x,z)\ \{R_{Or2}\}}\,\text{NA}$$

4.2.2 A partial specification for `MoveOr3`

Proof 2 (Proof scheme for `MoveOr3` (first part of Lemma 12))**.** *We keep here the notation introduced for the first proof 1. We use* (NA) *to separate the left and the right part of the* $|$*, then the right part has been done previously for the first proof, and the left part is very similar. We use once* (Cons) *with* $\square(Q_{Or3} \Rightarrow Q_{Or2})$ *and* $\square(R_{Or2} \Rightarrow R_{Or3})$ *We use the lemma 12 with the invocation rule to conclude.*

$$\dfrac{\dfrac{\dfrac{\text{Using Lemma 1 but not reproving it}}{\{Q_{Or2}\}\ \mathit{MoveOr2}(x,y,z)\ \{R_{Or2}\}}\,\text{IK}}{\{Q_{Or3}\}\ \mathit{Move}(x,y)\ \{R_{Or3}\}}\,\text{Cons} \qquad \dfrac{\dfrac{}{\{\varphi(x,w)\}\ \mathit{Move}(x,w)\ \{\mathit{On}(x,w)\}}\,\text{EF}}{\{Q_{Or3}\}\ \mathit{Move}(x,w)\ \{R_{Or3}\}}\,\text{Cons}}{\{Q_{Or3}\}\ \mathit{MoveOr2}(x,y,z) \,|\, \mathit{Move}(x,w)\ \{R_{Or3}\}}\,\text{NA}$$

4.2.3 A partial specification for `MoveThen`

Proof 3 (Proof scheme for `MoveThen`)**.** *Now, we have to use the rule* (Seq) *to separate* $\delta_1; \delta_3$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\{Q_{Or2}\}\ \mathit{MoveOr2}(x,y,z)\ \{R_{Or2}\}}{?!?}}{\{Q_{Or2} \wedge inv\}\ \mathit{MoveOr2}(x,y,z)\ \{R_{Or2} \wedge inv\}}\,?!?}{\{Q_{Then}\}\ \mathit{Move}(x,y)\ \{P\}}\,\text{Cons} \qquad \dfrac{\dfrac{}{\{\varphi(x,w)\}\ \mathit{Move}(x,w)\ \{\mathit{On}(x,w)\}}\,\text{EF}}{\{R_{Or2} \wedge P\}\ \mathit{Move}(x,w)\ \{R_{Then}\}}\,\text{Cons}}{\{Q_{Then}\}\ \mathit{MoveOr2}(x,y,z); \mathit{Move}(x,w)\ \{R_{Then}\}}\,\text{Seq}$$

Therefore, the precondition $\varphi(x,w) \subset Q$ *have to be available as a precondition for* δ_3*, and so the rule* (Cons) *shows that we need to deduce the conservation of* inv from *the proved specification of* `MoveOr2`*:*

$$\{Q_{Or2} \wedge \texttt{Clear}(x) \wedge (x,y,z \neq w) \wedge \texttt{Clear}(w)\}\quad \texttt{MoveOr2}(x,y,z)\quad \{R_{Or2} \wedge \texttt{Clear}(x) \wedge (x,y,z \neq w) \wedge \texttt{Clear}(w)\} \quad (4.5)$$

which looks like a frame axiom for the conserved property $\texttt{Clear}(x) \wedge (x,y,z \neq w) \wedge \texttt{Clear}(w) \stackrel{\text{def}}{=} inv$*, and with an effect part for* $\{Q_{Or2}\}\ \texttt{MoveOr2}(x,y,z)\ \{R_{Or2}\}$ *for the non-primitive procedure* `MoveOr2`*. This Liu-triple says that with* $x,y,z \neq w$*, if* $\texttt{MoveOr2}(x,y,z)$ *can be performed, it*

does not change the facts $\texttt{Clear}(x) \wedge \texttt{Clear}(w)$*, i.e. it says that* δ_1' *does not touch to a* ***disjoint*** *(*$x, y, z \neq w$*)* ***part of the domain*** *(in this case,* $\texttt{Clear}(w)$*).*

But, if we try to prove this without using the body of `MoveOr2`*, we see that no rule can be applied. Indeed, the only rules permitting to show the conservation of a property are* (FF)*, but only for properties of the form* $\overrightarrow{u} \neq \overrightarrow{v}$ *(i.e. without any fluents), and* (EF)*, which is only available for* ***primitive actions*** *A. So, the only way to prove* (4.5) *is to use the body of* `MoveOr2`*, as in proof 1, and to use twice the Frame Axiom* (EF) *with* $F = \texttt{Clear}$ *and* $\overrightarrow{X} = w$ *or* $\overrightarrow{X} = x$ *for* δ_1 *and* δ_2*.* □

So, as we said before, we can prove a specification for `MoveOr3` without using its body, because the two parts of its body (δ_1' and δ_3), are separated with a "`|`" and are disjoint enough to be treated separately. `MoveOr3` differs from `MoveThen` by having two parts separated with a "`;`" – so they are not really separated: δ_1' is executed *after* δ_3.

4.3 Diagnostic

This lack of modularity comes from the solution against the Frame problem used in `HG`. In [Liu02], Liu used the solution proposed by Reiter ([LR97, Rei01]), which consists of adding one *Effect and Frame Axiom* for every fluent F, where case-analyzes over the **primitives** actions are allowed (of the form $\left(a = A(\overrightarrow{x}) \wedge \dots\right)$). Therefore, Reiter's solution to the frame problem works well as long as we use only primitives actions, like in the proof of `MoveOr2`. But when we define intermediate procedure,we want to be able to specify it and to prove it in a natural way by giving Liu triples **only** for effects.

Hence, when we want to intermediate procedures them to write more complicated programs, we pointed out that this approach is not sufficient to describe the non-effects of programs (*i.e.* to face the *frame* problem): indeed in our example we encountered difficulties to prove that $\texttt{MoveOr2}(x, y, z)$ preserve the property $\texttt{Clear}(w)$ when $x, y, z \neq w$.

4.4 A solution for this lack of modularity?

We can describe effects and non-effects with the (EF) axioms, but only for the **primitive actions** A, because the Φ_F are written when the user axiomatizes her dynamic world, *i.e.* when she provides the primitive actions as the smallest component of `GOLOG` programs for this world. But axiomatize the world and use it to write procedure have to be a disjoint process. Thus, this mean that we **cannot** propose to update all the Φ_F when we define a new procedure (like `MoveOr2`). Moreover, propose to change each Φ_F implies to prove these modifications, so its requires (in the worst case) to prove one triple per fluents, for every new `GOLOG` procedure. Therefore this idea is not elegant, and in practice it will not scale.

On the other hand, in this example, we could require to extend the Lemma 1 and embed something like a Effect and Frame Axiom (rule (EF)) for the new procedure `MoveOr2`. We can ask to prove two axioms for every fluent F: one for the effects of the new procedure on F and another for the effects on $\neg F$. In fact, if we prove a frame axiom for each fluent and for every new procedures, it will be sufficient to *propagate* properties where it is needed (like in proof 3 when we asked `MoveOr2` to conserve $\texttt{Clear}(w)$). But here again this possibility is not applicable, because it requires to prove $2\sharp F$ just to define and specify one new procedure.

Therefore, our simple example of GOLOG procedures pointed out a serious lack of compositionality in the HG proof system, and in fact a real weakness of the situation calculus against the frame problem. This weakness is mainly due to Reiter's solution to the frame problem ([Rei01]), because writing Frame Axioms with a case-by-case analysis over actions can makes the update of an axiomatization laborious, especially when describing the non-effects of the form "$F(\overrightarrow{t})$ is still true after applying $A(\overrightarrow{x})$ if $F(\overrightarrow{t})$ was true before applying it and if $A(\overrightarrow{x})$ does not modify $F(\overrightarrow{t})$".

5 A more concrete language and a planning solver

We have presented the two domains of AI planning and formal verification, and the associated formalisms: the situation calculus, GOLOG and Liu's proof system. Now we will focused on our contribution about planning. We will present the STRIPS language as a subset of situation calculus, and shortly introduce the tools we used to make automated simulations, designed to show some limitations of the tool and the formalism. And finally we present some ideas of solutions, partly proven correct and efficient.

5.1 STRIPS as a subset of the situation calculus with a type system

The STRIPS formalism, originally developed at Stanford, by Fikes and Nilsson [FN72], is also a language to describe changing world, but it has the advantage of being more concrete than the situation calculus. The simplest way to view the STRIPS language, and its norm (PDDL), is to consider it as a zeroth-order restriction of the situation calculus, with the possibility to use a simple type system (also zeroth-order types, often called **sorts**).

We can use *types* for the sort `object`, as `block` for example, and then we give a type for every *objects* and *constants*, as 3 blocks `A`, `B` and `C`. This type system is often called a "tagging type system", because the declaration of typed objects consists at giving one label for each objects and this is written as follow: `A B C - block`.

Now we require predicates to be *typed*, like functions in functional languages such the λ-calculus or OCaml, or like procedures in the C language. For instance, with a C-like formalism, the relational fluents for the block world become:

$$\texttt{Clear(block b1)},\ \texttt{On(block b1, block b2)},\ \text{and}\texttt{OnFloor(block b1)}. \tag{5.1}$$

In STRIPS a *state* is a collection of facts, known to be true. There is a huge *restriction* made here: the situation calculus allow to add domain constraints as first-order formulas in Γ_0, but the STRIPS language can just consider zeroth-order facts. Therefore, if a state is a database of knowledge about the world, we have to clarify that a fact not present in this *database* is assumed as false. Again, the only difference made by STRIPS when describing actions is to add the tagging type system: now an action is declared with a typed signature. For instance, with a C-like formalism, the actions for the block world become:

$$\texttt{Move(block b1, block b2)},\ \text{and}\texttt{PutFloor(block b1, block b2)}. \tag{5.2}$$

Using STRIPS and not the situation calculus means we can no longer use undeclared variables, *i.e.* we can no longer write $\Pi_{\texttt{PutFloor}}(x) \stackrel{\text{def}}{=} \neg\exists y.\, \texttt{On}(y, x)$. This is a huge limitation, because we have to adapt its signature to contain each variables used to describe its precondition or its effects. For example, that is why we had to add a second argument to `PutFloor`, to represent the block `b2` on which `b1` is. We no longer use Π_A to now use $Prec(A)$. For the block worlds, the precondition can be rewritten as follow:

$$\Pi_{\texttt{Move}}(b1, b2)) \quad \equiv \quad \texttt{Clear(b1)} \wedge \texttt{Clear(b2)} \wedge \texttt{OnFloor(b1)}. \tag{5.3}$$

So, states are sets of zeroth-order formulas describing the world, and actions can be applied to transform a state into another. Another difference between the situation calculus and STRIPS is the point-of-view adopted to describe the effects of an action.

We already presented the intrinsic duality hidden here (*c.f.* remark 3), and `STRIPS` uses one formula for **each action** when the situation calculus require one axiom for each fluent. With our block example, we expect the action `Move` to actually move the block `b1` on top of `b2`. So, performing this action in a state Γ will change it by **adding** the fact `On(b1, b2)`, and this is done by writing the following **positive postcondition**:

$$Post_{add}(\texttt{Move(b1, b2)}) \equiv \texttt{On(b1, b2)}. \tag{5.4}$$

But, in Γ there will still be `OnFloor(b1, l)`, with some `l`, and `Clear(b2)`. And, remember, a state Γ is a **coherent** collection of fact, that means with no contradiction in it, so **it is not acceptable** to have both `On(b1, b2)` and `Clear(b2)` in the same time. This shows that we need to be able to **remove** facts from the database, when performing an action. This is done by adding *negative literals* in the postcondition, *i.e.* by writing a **negative postcondition**, like this one:

$$Post_{delete}(\texttt{Move(b1, b2)}) \equiv \neg\texttt{Clear(b2)} \wedge \neg\texttt{OnFloor(b1)}. \tag{5.5}$$

A postcondition is a conjunction of *positive literals*, called the **add list** and a conjunction of *negative literals*, called the **delete list**. Thus, we have to impose a condition on the delete list, because performing an action cannot result as deleting facts in our database Γ that were not there before. So, a postcondition can use a negative literal **iff** it was present in the precondition of the action. To continue with our example, this is indeed the case for the action `Move`, described by the positive post (5.4) and the negative post (5.5). Here, either we keep using two statements $Post_{add}$ and $Post_{delete}$, or we use just one $Post$, defined like this:

$$Post(a) \stackrel{\text{def}}{=} \left(\bigwedge_{f \in Post_{add}(a)} f\right) \wedge \left(\bigwedge_{f \in Post_{delete}(a)} \neg f\right), \tag{5.6}$$

with the condition that all the f are *positive literals*, which used only variables that are argument of a.

The description of an action is like a Hoare triple Now, we can describe an action $a(x_1, \ldots, x_n)$ by its precondition $Prec(a)$ and its postcondition $Post(a)$. This is like a Hoare triple: $\{Prec(a)\}\ a(\overrightarrow{x_i})\ \{Post(a)\}$. Remember that in the situation calculus, we do not use $Post(a)$ to describe effects, and thus it is normal that such triples did not appear as axioms for the proof system `HG`.

Then, we said that a can be applied in a state Γ if from Γ_i we can deduce $Prec(a)$, with a certain instantiation I of $\overrightarrow{x_i}$ (*i.e.* each x_i is mapped by I to an *object* or a *constant* y_i defined in the domain), and this is written $\Gamma_i \models_I Prec(a)$.

So, if $\Gamma_i \models_I Prec(a)$ then applying a change Γ_i to Γ_{i+1} where we **add** to Γ_i the facts in a's **add-list** (*i.e.* $Post_{add}(a)$) and **remove** the facts in a's **delete-list** (*i.e.* $Post_{delete}(a)$) :

$$\Gamma_{i+1} \stackrel{\text{def}}{=} \Gamma_i \cup Post_{add}(a(\overrightarrow{y}))\backslash Post_{delete}(a(\overrightarrow{y})). \tag{5.7}$$

This define the transition relation $\overset{a,I}{\rightsquigarrow}$, for database Γ which can be read as "applying the action a, with arguments given by the interpretation I"; and the function *Apply* by $Apply(Post(a, I), \Gamma_i) \stackrel{\text{def}}{=} \Gamma_{i+1}$.

A **planning domain** is the axiomatization of a dynamic world, and it is a declaration of *types*, *constants* tagged with types, *predicates* and *actions*, with typed signatures.

A **planning problem** is a declaration of a *domain*, a possibly empty list of *objects*, tagged with types, an initial database of facts Γ_0, and a goal Γ_{goal}. *Solving* this problem mean finding a sequence[7] $(a_j, I_j)_{0 \leqslant j \leqslant n}$ of actions a_j, instantiated by an interpretation I_j, such that $\Gamma_{goal} \equiv \Gamma_n$ and $\forall j. \Gamma_j \overset{a_j, I_j}{\rightsquigarrow} \Gamma_{j+1}$, with the *transition relation* $\overset{a, I}{\rightsquigarrow}$ defined in (5.7).

A solution to the first planning problem we introduced can be automatically generated, with the tool we used for our simulation, the only difference is the syntax used to represent this **path**, here we use the *predicate calculus* (*i.e.* classical zeroth-order logic), and the tool will produce a solution written in a *Lisp*-like syntax.

A complete axiomatization of the block world The appendix B presents the complete axiomatization for this basic block world, written with the `PDDL` syntax. It also includes two problems, figure 16 defines the first goal (`A` on `B` on `C`), and figure 17 defines the second goal (`A` on `B` on `C` on `A`).

`BNF` grammar A more syntactical approach could be done, but it is not very interesting to simply give here a grammar used to described domains and problems. The `STRIPS` language has been developed and used since 40 years, and nowadays an "official" version of this language has been proposed for the `IPC` competition. The result of this norm is a very clean `BNF` grammar. If the reader is curious about it, see [Kov11].

5.2 Tools used for our simulations

All the experiments were made on my personal machine, a `64-bits` PC running on `Ubuntu 12.04`, without parallelization, so with one core, running at `3.2 GHz`, and with `2 Go` of `Ram`.

In verification, different kinds of tools exist, and in particular: proof inference tools (designed to find a prove for a program, like `Z3` or a type for a program like the in `OCaml`) and proof checkers (designed to verify a given proof and possibly give informations to help the programmer to fix it, like a prove assistant like `Coq`). In `AI`, theorem prover becomes planning solver (also called *planner*), and proof checker becomes plan validation tool (also called *validator*).

A planner: `pyperplan` To solve the planning problem, with a domain and a problem written in `PDDL`, we used the state-of-the-art solver `pyperplan` (see [Mal11] for more details).

`pyperplan` is a lightweight `STRIPS` planner written in `Python 3`, developed for the planning practical course at *Albert-Ludwigs-Universität Freiburg* during the winter term 2010/2011 and is an open-source software. It supports the following `PDDL` fragment: deterministic and **positive**[8] `STRIPS` without action costs, and without domain constraints.

Basically, `pyperplan` use two files: `domain.pddl`, defining the domain (types, predicates, actions) and `task.pddl`, defining the planning problem (objects, initial and goal state). For an example of such files, see the appendix B. So, with this two files, `pyperplan` try to solve

[7] This sequence can be empty, if $\Gamma_0 \vDash \Gamma_{goal}$, $n = 0$. So in general, $n \geqslant 0$.

[8] See section 5.3.1 for an explanation of this term.

the path-finding problem. If there is a solution, it will be written to `task.pddl.soln`, in the *Lisp*-like syntax; and if there is none, a warning message is printed.

A validator: `validate` (VAL) Even if `pyperplan` seems to be sure[9] of the solution it gives, it allows the use of an external *path validation tool*. We used the state-of-the-art `validate` tool. This plan validation software is also freely available and open-source (see [LF11] for more details).

5.3 Sub-language accepted by `pyperplan`

Here we present some experiments we made with the tool `pyperplan` and the `STRIPS` formalism. Those experiments were especially designed to point out some limitations of the tool, but also a **few weaknesses of the language**.

5.3.1 Positive-`STRIPS` seems to be a serious limitation

One limitation `pyperplan` that it accepts only positive-`STRIPS`. This sub language does not allow to use a negative literal ($\neg P(\overrightarrow{x_j})$) in a *state* Γ_i; nor in the *goal* Γ_{goal}; neither in an action's *precondition*; nor in an action's *delete-list postcondition* ($Post_{delete}$) which is not present in its positive form in the precondition (*i.e.* the *delete-list* can only delete facts explicitly assumed as true by $Prec$).

Let us consider the classical **Yale Shooting** problem. We consider two types, `man` and `gun`, two predicates, `Loaded(gun)` to say that a gun is ready to shoot and `Alive(man)` to say that the man is still alive, and two actions `Shoot(man, gun)` to kill the man with a loaded gun, and `Load(gun)` to load the gun. An axiomatization in general-`STRIPS` could be $Prec(\texttt{Load}(g)) \stackrel{\text{def}}{=} \neg\texttt{Loaded}(g)$, $Prec(\texttt{Shoot}(m,g)) \stackrel{\text{def}}{=} \texttt{Alive}(m) \wedge \texttt{Loaded}(g)$ for preconditions and $Post(\texttt{Load}(g)) \stackrel{\text{def}}{=} \texttt{Loaded}(g)$, $Post(\texttt{Shoot}(m,g)) \stackrel{\text{def}}{=} \neg\texttt{Alive}(m) \wedge \neg\texttt{Loaded}(g)$ for postconditions.

Now let try to axiomatize it *naively* in positive-`STRIPS`. For the action `Load`, if we have a predicate `Loaded(g)`, it **has to be** in negative form in $Prec(\texttt{Load})$, and this is not allowed. Otherwise, if we have a predicate `NotLoaded(g)`, we can put it in positive-form in $Prec(\texttt{Load})$, but for $Post(\texttt{Load})$, we cannot use $\neg$`NotLoaded(g)`.

Whereof, this is a **huge limitation**, and an example as simple as the Yale Shooting problem *cannot be naively axiomatized* in positive-`STRIPS` (at least, without adding other predicates or other types). We have a systematic and efficient solution for this limitation, presented in section 6.2.

5.3.2 Updating an axiomatized world is not so easy

Let us imagine a *colored block world*, defined with the type `color`; objects of type `color` like `red`, `blue` or `yellow`; a predicate `Color(block, color)` giving the current color of a block;

[9] This correctness depends on which heuristic (option `-H`) and search algorithm (option `-s`) is used, an inefficient but sure choice is to use a breath-first search and no heuristic, to at-least by theoretically sure of the given solution (only theoretically, we never are certain of the absence of bugs in the solver!).

and one action `Paint(cu, co, co')`, with $Prec$(`Paint`) = `Color(cu, co)`, $Post_{add}$(`Paint`) = `Color(cu, co')`, $Post_{delete}$(`Paint`) = `Color(cu, co)`. If we want to join this colored block world with the previously defined block world, we add the facts `Color(A, red)`, `Color(B, blue)` and `Color(C, yellow)` to the initial fact database Γ_0.

Then, there is no problem, we can move blocks (using `Move` or `PutFloor`) without having to consider their colors and we can paint them (with `Paint`) without having to consider their relatives positions.

But if now we want to say that the floor is a huge paint tray, of one color (`red` for example, written with the new predicate `ColorFloor(color)`), we would like to be able to update the action `PutFloor`, because putting a block to the floor will make it become `red`. So, for this "new version" of `PutFloor`, we **have to** add the following fact in its precondition: `ColorFloor(co')`, to say that `co'` is the color of the floor, and `Color(b1, co)`, to be able to delete this fact in the $Post_{delete}$ (remember, we require to delete only facts present in the precondition);

And we also have to update both the positive and negative postconditions: we add to $Post_{add}$: `Color(co')`, the block takes now the color of the floor; we add to $Post_{delete}$: `Color(b1, co)`, to delete the old color, One of the limitation of `PDDL` is the absence of free variable in $Prec(a)$ and $Post(a)$.
Here, we used `co'` to be the floor's color, and `co` to be `b1`'s color. So, we **have to change the signature** for this action, and it becomes `PutFloor(?b1 ?b2 - block ?co ?co' - color)` (in the *Lisp*-like syntax).

This simple experiment showed that if it is done naively, adding 2 new elements in the block world (block color and floor color) requires to re-write all the moving actions. So this approach requires to change some previously defined actions (the old `Move` for instance). Indeed, this new version of `PutFloor` does not have the same signature, so everything we have proven with the first version is not valid anymore. Thus, the main point to keep in mind is that the basic way to update an axiomatization is not very efficient, and **it is not modular**. The question of how to easily update en existing axiomatization is in-fact related the question of **procedural modularity**, already discussed before.

The duality *predicate-action* In the in remark 3, we presented a duality between effect axioms (EF) (one Φ_F for each F, the choice made in the situation calculus) and action postcondition (one $Post(a(\overrightarrow{x}))$ for each a, the choice made in `STRIPS`). We use this paragraph to remark that each choice is *strong against one kind of update and weak against the other*:

(EF) is efficient when we add a new fluent F', we just have to write a new $\Phi_{F'}$, but is weak when we add a new action a', because in the worst case we have to change each previous Φ_{F_j} to add a case for a'.

Post(a) is efficient when we add a new action a', we just have to write a new $Post(a')$, but is weak when we add a new fluent F', because in the worst case we have to change each previous $Post(a_j)$ to express their effects on F'.

5.4 `pyperplan` does not scale well?

In the `STRIPS` point-of-view, there is a *frame hypothesis*, implicitly given when we described the relation $\overset{a,I}{\rightsquigarrow}$. And in the situation calculus point-of-view, this hypothesis is explicitly given

by the Effect and Frame Axioms. Applying an action $a(\overrightarrow{x})$ only change the knowledge set Γ by adding the facts in $Post_{add}(a)$ and removing the facts in $Post_{delete}(a)$. So, a fact $F(\overrightarrow{y})$ not concerned by $Post(a)$ will *not* be modified by a.

This frame hypothesis is already a good idea, because for planning it works, and the `STRIPS` formalism is relatively concise thanks to this hypothesis. But it is not enough to scale when we describe large worlds, and it is not enough to have a good modularity. For example, if we want to describe a robot's actions for the first floor and the second floor of a building, we would like to say that *every things* happening in the two floors are disjoint. Therefore, for planning, if the initial state and the goal state does not include facts about the second floor, the planner should be able to find a plan only in the first floor world. But will it be able to do it as efficiently as in there was only one floor? The benchmarks presented below show that it is not the case.

About benchmarking `pyperplan` We present here some of the benchmarks we made, with the solver `pyperplan`. This simulation was made with the block world, axiomatized a little differently that the one included in the appendix: this version uses 4 actions and 4 relational fluents. All the files used for the simulations are available on-line, on my web page http://www.dptinfo.ens-cachan.fr/~lbesson/stageM1/benchmarks/. We used the following `GNU Bash` command to time the simulation: `time pyperplan.py -H hff -s gbf domain.pddl task.pddl`.

First experiment: increasing the number of objects We consider one robot, with the goal of building a tower of n blocks (*i.e.* $\Gamma_{goal}(n) \stackrel{\text{def}}{=} \texttt{On}(\texttt{B}_n, \texttt{B}_{n-1}), \dots, \texttt{On}(\texttt{B}_2, \texttt{B}_1)$). And we present in the next array the time needed by `pyperplan` to solve a planning problem defining more and more objects. The file used are `blocks/task*.pddl`.

Goal file	Number of objects n	Computation time (in s)	Lenght of the plan	Optimal plan
task01	4	0.130	6	6
task13	8	1.880	34	14
task33	16	493.097	168	30
task35	17	Still running!	No plan found yet	32

Figure 8: A goal more and more complicated in the same world.

This simulation (figure 8) shows that increasing the complexity of the goal increases the exponentially the time required to solve the planning problem. And the length of the plan discovered by `pyperplan` grows also exponentially, even if the length of the optimal plan grows linearly (with n objects, there is a solution of length $2(n-1)$ for $\Gamma_{goal}(n)$ in this second version of the block world). So, for too sophisticated or too big goals, the tool is not usable; in other words, it does not scale very well.

Second experiment: cross-product of disjoint worlds We consider now different robots, each operating on a different table. Each pair (robot, table) is a disjoint block world. This is like making a Cartesian product of vectorial sub-space in linear algebra: each sub-space

operates independently of the others, but they are considered unified in one big vectorial space. The file used are in the subdirectories `blocks`, `block_double`, `block_triple`, and `block_4`.

Goal file (*.pddl*)	Number of objects	Computation time (in s)	Lenght of the plan	Optimal plan
task01	4	0.130	6	6
task04-fixedgoal	5	0.133	6	6
task35-fixedgoal	30	0.149	6	6

Figure 9: Growing number of objects but constant goal in the same world.

Number of object	Number of world	Computation time (in s)	Lenght of the plan	Optimal plan
4	1	0.130	6	6
4,0	2	0.133	6	6
4,0,0	3	0.135	6	6
4,0,0,0	4	0.137	6	6

Figure 10: Disjoint-union of similar worlds but constant goal.

This second experiment (figure 10) shows that increasing the complexity of the world (*i.e.* the number of actions and fluents) but with keeping the same simple goal (a tower of 4 blocks) does not change very much the computation time. In fact, what changed is just the time needed by `pyperplan` to read the files. However, adding objects, even if they are not used in the goal (figure 9), slightly increases the difficulty of the problem, because the solver may consider actions applicable to useless objects, and this can be time-consuming.

Number of object	Number of world	Computation time (in s)	Lenght of the plan	Optimal plan
4,0,0,0	4	0.137	6	6
4,4,0,0	4	0.277	24	12
4,4,4,0	4	1.213	42	18
4,4,4,4	4	3.486	60	24

Figure 11: Disjoint-union of similar worlds and growing goal.

This last array (figure 11) shows that increasing the complexity of the world (*i.e.* the number of actions, fluents, and objects) and of the goal in a disjoint way makes the computation time explodes.

We presented here a few experiments, but we have made many others, mainly with more sophisticated dynamic world. For example, one of the more impressive simulated world was a freecell solitaire game. What is important to remember is that the tool `pyperplan` does not scale very well when it considers large worlds, and it use not efficiently the fact that some sub-parts of the domain are disjoint.

6 Techniques to fix some weaknesses

6.1 Working without the "tagging" type mechanism

`blackbox`, the first planner we used was more minimalistic, and it didn't support the `STRIPS` `typing` extension. This extension is used to declare somes **types** (like `gun` and `man` for the Yale Shooting problem, or `block` and `color` for the block world). From the first introduction of the `STRIPS` syntax, we used types to write the signature for predicates and for actions. Those types as the simplest types we could consider, and are usually referred as *tagging* types (types are like labels or tags for objects).

For example, we also could axiomatize the Yale Shooting problem without using types. For instance, in `STRIPS` without types, we can add two *fresh* predicates $\texttt{IsOfType}_{gun}$ and $\texttt{IsOfType}_{man}$; and in Γ_0, we add `IsGun(Colt)`, and `IsMan(Joe)`. And in $Prec(\texttt{Shoot}(m, g))$, we require $\texttt{IsOfType}_{gun}\texttt{(g)}$ and $\texttt{IsOfType}_{man}\texttt{(m)}$.

This **emulated type system** requires to add one predicate $\texttt{IsOfType}_{type}$ for each types *type*, to add one fact $\texttt{IsOfType}_{type}\texttt{(o)}$ in Γ_0 for each objects `o` previously defined with `o - type`. And for each action a of arity n, we have to add n typing fact in $Prec(a)$. When we consider the length of the formulas and the size of the `PDDL` files, this *emulated type system* is equivalent to give a's signature in the first form $a(?x_1 - t_1 \ldots ?x_n - t_n)$.

But in fact, the type system, emulated or not, seems to not be needed. We conjecture that we can remove it, and hence only use an **implicit** type system. In the Yale Shooting example, in $Prec(\texttt{Shoot(m,g)})$, we have `Loaded(g)` and `Alive(m)`, so this imply that `g`'s implicit type is the type accepted by `Loaded` (which is `gun`) and `m`'s implicit type is the type accepted by `Alive` (*i.e.* `man`). Of course, Γ_0 have to be sound according to this implicit type, *i.e.* it could be possible for a human programmer to produce an axiomatization in the system with the type system by simply add types where they are needed, without producing anything unsound.

In practice, `pyperplan` allows to use the typing extension, but also allows not to use it; thus the extension is never truely needed, because we can **simulate** it. And even without simulate a type system, if the programmer is aware of the implicit type system, we think that we can completely remove it, without changing the expressiveness of the language, but without really improving the conciseness of the descriptions (using only the implicit type system allows to shorten each action's signature by something like $c \times n \times \max \texttt{length}(type)$, where n is the action's arity, and c is a certain small constant).

6.2 Truth values reification, or how positive-`STRIPS` is equivalent to general-`STRIPS`

We previously said that `pyperplan` only accepts **positive**-`STRIPS`, and we present here a solution to fix this limitation. For the Yale Shooting example presented in the section 5.3.1, we did not know if we had to use the predicate `Loaded` or `NotLoaded` because in fact none of them was a good approach.

First, we introduce, in every domains, a new type: `bool`, representing boolean values, and two constants[10] `true` and `false` of type `bool`. And then, for every relational fluent

[10] Constants are defined in the field `:constants` if it is allowed in the domain file (`domain.pddl`), otherwise we

`F`$(y_j - t_j)$ defined in the domain, we change it to a *fresh* new fluent `ValueF`$(y_j - t_j, b - bool)$, with an extra boolean argument `b`. This new fluent is interpreted as `ValueF`$(y,$`true`$) \equiv$ `F(y)` and `ValueF`$(y,$`false`$) \equiv \neg$`F(y)` Whenever a literal `F(y)` is used in a $Prec($`a`$)$, we change it to `ValueF(y, true)`, and a negative literal $\neg$`F(y)` is changed to `ValueF(y, false)`. This little transformation increases the length of $Prec(a)$ by a small constant c_1 (in the worst case, transforming $\neg$`F(y)` into `ValueF(y, false)`, so $c_1 \leqslant 15$)

And then we update $Post(a)$ as follow: if `F(y)` is changed from being true to false by a (resp. from false to true), we add `ValueF(y, false)` in $Post_{add}(a)$ (resp. `ValueF(y, true)`) and we add `ValueF(y, true)` in $Post_{delete}(a)$ (resp. `ValueF(y, false)`).

This technique shows that in term of expressiveness, positive-`STRIPS` is equivalent to `STRIPS`. And it is also the case in term of *conciseness*. Indeed this second transformation increases the length of $Post(a)$ by multiply it by another small constant c_2: here we transform $\neg$?`F(y)` into $\neg$?`ValueF(y, true)` $\wedge$ $\neg$?`ValueF(y, false)`, so $c_2 \leqslant 3$.

However, in term of *simplicity*, this **truth values reification technique**[11] is obviously not "nice" to work with. But these two transformations are simple enough to be automatically done by a tool; even if we did not take the time to look at the code of `pyperplan`. So, we can claim that this technique can be used in practice, without changing the performance too much[12]. And thus, we conclude that **positive**-`STRIPS` is almost **equivalent** to **general**-`STRIPS`.

6.3 Emulating the `STRIPS` `equality` extension

The `equality` extension allow to use the predicate =, to express that some objects are different or equals. Here we re-use the example of the suitcase, mentionned in the introduction (see section 1). We model a *suitcase* `s` with two different latches `L1` and `L2` (type `latch`). Such a latch can be up and down (`Up(l)` or $\neg$`Up(l)`), and the suitcase can be open or close (`Open(s)` or $\neg$`Open(s)`), and a latch can be related to a suitcase (`Related(s,l)`). We can open the suitcase **iff** its **two** latches are up. So the action `OpenSuitcase(s, L1, L2)` requires that `L1` $\neq$ `L2` (*i.e.* `(not (= L1 L2))` in the *Lisp*-like syntax).

There is a **simple** but **inefficient** way to use this simulate this extension in a subset of `STRIPS`. For every types *type* defined in a domain, we add a predicate `Eq`$_{type}$`(?x ?y - ty ?b - bool)`, using the boolean reification trick (introduced previously in section 6.2). In a precondition, we substitute `(not (= ?x ?y))` by `(Eq`$_{type}$ `?x ?y false)` and `(= ?x ?y)` by `(Eq`$_{type}$ `?x ?y true)` (in the `PDDL` syntax). In Γ_0, in the worst case we **have to** add `(Eq`$_{type}$ `?x ?y false)` for every different objects `x,y` of type *type*, and we might even have to add `(Eq`$_{type}$ `?x ?x true)` for every objects `x` of type *type* (we did not find example that requires it, but possibly there is one).

Consequently, this technique *works*, but is very inefficient because it could require to add ${n_{type}}^2$ facts in Γ_0 for every types *type*, and where n_{type} is the number of objects of type *type*.

add them in every problem file (`task.pddl`).

[11] In this case, **reification** means transform a concept (the truth value of a predicate) into an object (`true` and `false`).

[12] As in the study of algorithms's complexity, we are only interested in the order of complexity, multiplying the files's length by a small constant is acceptable.

7 Sum-up and perspectives

In this last section, we will briefly sum up what we have done in this report, then explain some directions we could have explored with more time. Finally, I will quickly give a short opinion about this internship.

7.1 Sum-up

In the first part, we started by presenting the block world, an example of a dynamic system, interpreted as a domain of exploration for a small robot. Dynamic systems such as this one are described by a higher order predicate language called situation calculus ([MH68, Rei01]). When one considers a robot *exploring* a domain, the question of reachability is natural, and in `AI` this is studied in a research field called planning.

But another natural question is to program the robot to control it in a more sophisticated way that just a sequence of orders. The `GOLOG` language ([LRL$^+$97]) is used to program such robots and more easily, and for this goal `GOLOG` provides a compositional way to define intermediate procedures and use them to increase conciseness and simplicity of other programs. This was illustrated by writing the first program `MoveOr2` and then by using it to define `MoveOr3` and `MoveThen`, to control the robot operating on the block world.

To formally describe this procedure `MoveOr2`, we introduced the notion of Liu triples, very similar to Hoare triples from classical Hoare logic, with the difference that Liu logic is used to write specifications for robot programs, *i.e.* in a programming language where the smallest components are user-defined action on a dynamic world axiomatized with the situation calculus.

To prove such specification, we used the `HG` proof system, also very similar to Hoare logic. The proof of a partial specification for `MoveOr2` is the first concrete contact with `HG`. We were mainly interested in proving or disproving procedural modularity for `HG`, as the Hoare proof system is thanks to separation logic ([Rey02]). Unfortunately, we found a loophole in `HG`, exposed with the counter-example `MoveThen`. To strengthen the importance of this counter-example, and expose the proof of `MoveThen` more didactically, we also wrote and proved an other program `MoveOr3`, very similar to the problematic one (`MoveThen`). Indeed, `HG` is not compositional in this example because the proof of `MoveThen` required to demonstrate some additional conservation lemma for the intermediate procedure `MoveOr2`.

In fact, the situation calculus provides a solution to the *frame problem* (*i.e.* the difficulty to describe the non-effects of actions) by writing *successor state axiom* for every fluents, of the form of a case-by-case analyzes over primitive actions. The proof system use these successor state axioms to infer Frame and Effect axioms (see the rule (EF)), and they are used to axiomatize both *effects* and *non-effects* on relational fluents, but only for primitive actions. As it is not possible to update these EF axioms when writing new `GOLOG` procedures, `HG` is not able to consider a proven procedure as an atom, and thus it suffers for a serious lack of compositionality.

A second part of the work done during my internship was more concrete and more related to the question of planning. Historically, during the first weeks we were mainly focused on this aspect, and we became interested in the `HG` proof system in the last third of my internship. But

it was more didactic to first expose the situation calculus, GOLOG programs and to study here compositionality for HG.

Hence, we focused on a more concrete approach in the second part of this report. We choose to use a planning solver called pyperplan, partly because it was the more recent I found, the simpler to use, but mainly because the formalism used to describe the exploration space (*i.e.* the dynamic system where the robot work) is inspired from the situation calculus. pyperplan reads domain and problem files written in a *List*-like version of the STRIPS formalism, normalized by the *Internation Planning Competition* between 2002 and 2011 as the norm PDDL ([Kov11]). Basically, STRIPS is a subset of the situation calculus, without quantifiers, and with a labeling type system. The main difference comes from how the effects of actions are described: STRIPS prefers a more natural way, by writing positive and negative postconditions instead of successor state axioms as the situation calculus does.

Two concrete axiomatizations of a domain and a problem are embedded in the appendix (see sections A and B). We used them to point out some concrete limitations of the solver, thus we have done many different numerical simulations, and the most interesting results are explained in the section 5.4. The solver works well only for small worlds, because its exploration algorithm does not scale when the complexity of the goal increases. Another limitation of pyperplan comes when solving disjoint problems defined as one: the solver is not clever enough to find disjoint parts in the domain, therefore it suffers from a serious concrete weakness when dealing with worlds with disjointness.

This weakness is also present in the situation calculus, and in STRIPS. Historically, specifying disjointness was one of the problematics at the origin of the development of separation logic, which can be quickly presented as an extension of Hoare proof system with a new symbol and a new rule to explicitly express disjointness. We were focused on compositionality in this report, thus we pointed out both formal and concrete lack of compositionality in the situation calculus, the HG proof system, and the solver pyperplan and its formalism (STRIPS, PDDL). Therefore, a natural next step is to fix these limitations, and even if we did not have the time to really do this, we started to consider possible direction, as exposed in the next section.

7.2 Ideas, other questions and future works

We would now conclude and extend this digest by presenting one question we could also have worked on, and some ideas or direction we have not really studied by lack of time.

7.2.1 Termination for GOLOG procedures

We worked on verification for robot procedures, but we only consider one kind of specification: Liu-triple. Expressing termination as a first-order formula is not possible, therefore this formalism can not be used to work on termination. We did not take time to work on termination, mainly because it have already been studied. If we assume that every primitive actions terminates when they are applied to executable situations, then the only way to introduce non-termination in a GOLOG program is with the non-deterministic iteration (δ^*). It seems possible to convert a GOLOG procedure into a logical language such Prolog, and then use what is already available to try to proof its termination. Of course, we only expect to "try to" because GOLOG is a Turing-complete language and therefore, its termination is undecidable.

7.2.2 Adding a separation conjunction to express disjointness

The simulations presented in section 5.4 showed that `STRIPS` and `pyperplan` do not scale well when describing a world as a union of disjoint parts, although this particular case is frequent, thus fixing this weakness could be an iterating goal. Nor `STRIPS` neither `pyperplan` have a way to express disjointness, so introducing a way to specify disjoint for some subset of the domain (*i.e.* of the axiomatization) is a possible direction.

Another direction could be to interpret the block world by partial models, *i.e.* partial boolean functions on the block world ($f : \texttt{Blocks}^n \hookrightarrow \mathbb{B}$) as it is done in the domain of separation logic ([Rey02, CDOY11]). If we could find a way to convert an axiomatization (written in the situation calculus formalism or in `STRIPS`) to a list of separation logic specifications for the primitive actions. Then, it might be possible to simply use the extended Hoare proof system from separation logic ([HV13, O'H08]) to proof `GOLOG` programs, in a more compositional way than with `HG`, thanks to the Frame Rule of the separation logic. In the last two weeks, we started to consider this direction, with Peter O'Hearn, but by lack of time, we did not have worked more than that.

7.2.3 Nicely update an axiomatization with action refinements

As we said before (in section 5.3.2), one kind of compositionality appears when we want to complete or update an axiomatization. In this situation calculus point-of-view, one might have to re-write all successor state axioms when adding one new action (*i.e.* one per fluent).

And in the `STRIPS` point-of-view, one might have to re-write every postconditions when adding one new fluent. In fact, it more likely necessary change these axioms by adding something new rather than completely rewrite it.

With this idea in mind, one way to make this easier for the programmer could to have a concrete syntax to only write what changes. For instance, with the example of color back from section 5.3.2, we want to have two ways to put a block on the floor: to a color back or not. One way to generate these two versions of the action `PutFloor` could be to write the first one, as we did previously, and then to *refine* it to write the second one. Refining it, mean that we specify something to add in its precondition, and either specify some additional effects, either specify a completely new set of effects.

This idea is related to class inheritance as encountered in Object Oriented languages (*OOP*). Hence, one should first become familiar with the theory behind inheritance and try to apply it to actions refinement. We hope that this idea will ease the work needed to produce a suitable axiomatization, by shortening some primitive actions definition. We started to explore this direction, in particular with the work of Matthew Parkinson ([Par07]), but we preferred to stay focused on `HG` and its compositionality, by lack of time.

7.3 Personal conclusion

Even if this internship was short (10 weeks only), it was a very rewarding experience. First of all, the opportunity to live and work in a different city, in a different country, and to speak a different language is an enriching experience in many ways. Secondly, I enjoyed being included in a research team and working with interesting people on a daily basis. Finally, working on the

same questions during two months and a half was enriching, especially fundamental problems like the frame problem and modularity.

This first year M.Sc. internship was my second one, and I think it helps me develop some useful skills, as for instance scientific English, working with a tutor and in a research team, reading research papers more efficiently; or also practical skills like BibTeX for instance.

I might continue to study with Jules Villard and Peter O'Hearn some questions raised in this report, especially the future work exposed in section 7.2.2 might result as a significant solution to improve modularity, for both the situation calculus and HG. Such contribution would be a real breakthrough in this domain and would surely lead to publication. In this case, it would be greatly rewarding to be part of its development.

Thanks for the reading !

Please, do not hesitate to contact me for any question, any comment or to signal any problem about this internship report by email at Lilian.BESSON[at]ens-cachan[dot]fr or get more information on my web page.

Mark Jules and Peter gave me 19/20. I got the mark 17.7/20 from the Computer Science jury at ENS Cachan, and 18.50/20 from the Maths jury at ENS Cachan.

Acknowledgements

First, I would like to thank Jean Goubault-Larrecq, Hubert Comon, Serge Haddad (from LSV & ENS Cachan), Damien Vergnaud (from ENS Ulm) and Florian de Vuyst (from CMLA & ENS Cachan) for helping me to find an internship.

I would also like to thank the UCL, the CS department, the *PPLV* group, and especially Carsten Fuhs and Peter O'Hearn for their support and their warm hospitality.

And, most of all, I would like to thank Jules Villard for his constant support.

– End –

Bibliography

[CDOY11] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. *Journal of the ACM*, 2011. 2.4, 7.2.2

[FN72] Richard E. Fikes and Nils J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3):189–208, 1972. 5.1

[Hoa69] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969. 1.1, 2.3

[HV13] Aquinas Hobor and Jules Villard. The ramifications of sharing in data structures. 2013. 7.2.2

[Kov11] Daniel L. Kovasc. Bnf definition of pddl 3.1. Technical report, international planning competition (IPC), 2011. 5.1, 7.1

[LF11] Derek Long and Maria Fox, 2011. validate, an efficient and easy-to-use plan validation software, written in `C++`. 5.2

[Lin08] Fangzhen Lin. *Handbook of Knowledge Representation*, volume 1, chapter 16. Elsevier Science, 2008. 1.1, 1

[Liu02] Yongmei Liu. A hoare-style proof system for robot programs. 2002. 1.1, 3.3, 4.3

[LR97] Fangzhen Lin and Raymond Reiter. Rules as actions, a situation calculus semantics for logic programs. volume 31, pages 299–330, 1997. 1.1, 4.3

[LRL+97] Hector J Levesque, Raymond Reiter, Yves Lesperance, Fangzhen Lin, and Richard B Scherl. Golog: A logic programming language for dynamic domains. *The Journal of Logic Programming*, 31(1):59–83, 1997. 1.1, 2.2, 3.2, 7.1

[Mal11] Helmert Malte, 2011. pyperplan, a lightweight `STRIPS` planner, written in `python 3`. 5.2

[MH68] John McCarthy and Patrick Hayes. *Some philosophical problems from the standpoint of artificial intelligence*. Stanford University, 1968. 1.1, 7.1

[O'H08] Peter O'Hearn. Tutorial on separation logic (invited tutorial). In *Computer Aided Verification*, pages 19–21. Springer, 2008. 7.2.2

[Par07] Matthew Parkinson. Class invariants: The end of the road. In *International Workshop on Aliasing, Confinement and Ownership*, volume 23, 2007. 7.2.3

[Rei01] Raymond Reiter. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. The MIT Press, Massachusetts, MA, illustrated edition edition, 2001. 1.1, 1, 3.3, 4.3, 4.4, 7.1

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. 2002. 7.1, 7.2.2

A Path finding for graphs seen as a robot problem

We define here a planning domain and problem, to continue the analogy between path finding in graph theory and in AI. This analogy is almost fruitless, we did not use any results from graph theory to help planning in general, bu we found useful to presents a very simple dynamic world, based on the very well-known model of deterministic finite graphs.

This give an opportunity to see an axiomatization in STRIPS written according to the PDDL norm.

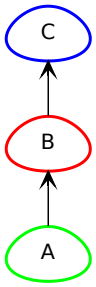
```
(define (domain GRAPH) ; A graph axiomatization in PDDL
(:requirements :strips :typing)
(:types edge explorer) ; One explorer visites the graph
(:predicates ; The explorer e is on s, or have visited s
       (IsOn          ?e - explorer ?s - edge)
       (Explored      ?e - explorer ?s - edge)
       (Vertex        ?x ?y  - edge)) ; There is an edge from x to y
(:action GoTo ; The explorer e move FROM s1 TO s2
     :parameters   (?e - explorer   ?s1 ?s2 - edge)
     :precondition (and (IsOn ?e ?s1) (Vertex ?s1 ?s2))
     :effect (and (not (IsOn ?e ?s1)) (IsOn ?e ?s2) (Explored ?e ?s2) ) ) )
```

Figure 12: A graph axiomatization (Vertex) and a graph explorer (On, GoTo) in PDDL.

We use the PDDL formalism to axiomatize a graph and a graph explorer, and then we use the tool pyperplan to solve a little path finding problem, with this axiomatization (the graph is explored by an object of type explorer, as a robot moving on the graph)

One example is showed here, for the very simple graph $\mathcal{G}_1 \stackrel{\text{def}}{=} (\{A, B, C\}, \{A \rightsquigarrow B, B \rightsquigarrow C\})$.

An achievable goal is to ask the explorer Dora to start from A, and to visit B.




```
(define   (problem REACHABLE)
(:domain  GRAPH)
(:objects A B C - edge Dora - explorer) ;   A->B   and   B->C
(:init    (IsOn Dora A) (Explored Dora A) (Vertex A B) (Vertex B C) )
(:goal    (and (Explored Dora C) ) ) )
```

Figure 13: $\mathcal{G}_1$ (in the left), and its axiomatization in PDDL as a reachable goal.

The solver pyperplan proposed the following straightforward solution:

```
(goto dora a b)
(goto dora b c)
```

Figure 14: An automatically generated solution for the problem defined by figure 13.

B A simple axiomatization of the block world, in the PDDL syntax

Here are included 3 `pddl` files.

The first one (`domain.pddl`) defines the block world domain, used in all this report. Those files are written accordingly to the latest norm (`PDDL v3.1`), in the *Lisp*-like syntax, with the `typing` extension, and in positive-`STRIPS` only, so they can be used as-provided by the solver `pyperplan`.

```
(define (domain BLOCKS)
(:requirements :strips :typing)
(:types block)
(:predicates
       (On          ?x ?y  - block)
       (OnFloor     ?x     - block)
       (Clear       ?x     - block))
(:action Move
     :parameters   (?block1 ?block2 - block)
     :precondition (and (Clear ?block1) (Clear ?block2) (OnFloor ?block1))
     :effect (and (not (Clear ?block2)) (not (OnFloor ?block1)) (On ?block1 ?block2) ) )
(:action PutFloor
     :parameters   (?block1 ?block2 - block)
     :precondition (and (Clear ?block1) (On ?block1 ?block2))
     :effect (and (OnFloor ?block1) (Clear ?block2) (not (On ?block1 ?block2)) ) ) )
```

Figure 15: The block domain.

The second one defines the achievable goal presented in figure 1 (a tower, `A` on `B` on `C`).

```
;;; This goal is reachable, and the solver proves his reachability, by giving a solution
(define  (problem REACHABLE)
(:domain  BLOCKS)
(:objects A B C - block)
(:init    (Clear A) (OnFloor A) (Clear B) (OnFloor B) (Clear C) (OnFloor C) )
(:goal    (and (On A B) (On B C) ) ) )
;;; The generated solution  is: (Move B C) (Move A B)
```

Figure 16: A reachable goal for the block world.

And the last one defines a similar goal, also presented in section 2.1, but non-achievable (`A` on `B` on `C` on `A`).

```
;;; This goal is not reachable, and the solver proves his non-reachability.
(define   (problem NON-REACHABLE)
(:domain  BLOCKS)
(:objects A B C - block)
(:init   (Clear A) (OnFloor A) (Clear B) (OnFloor B) (Clear C) (OnFloor C) )
(:goal   (and (On A B) (On B C) (On C A) ) ) )
```

Figure 17: A non-reachable goal for the block world.