Guidance of a Refinement-based Acting Engine with a Hierarchical Temporal Planner

Jérémy Turi, Arthur Bit-Monnot

Guidance of a Refinement-based Acting Engine with a Hierarchical Temporal Planner

Jérémy Turi, Arthur Bit-Monnot
LAAS-CNRS, Université de Toulouse, INSA, CNRS, Toulouse, France
jeremy.turi@laas.fr, abitmonnot@laas.fr

Abstract

Endowing robotic platforms with advanced deliberation capabilities to support autonomous and intelligent behavior has been a key objective of the scientific community for several decades. Where many acting systems propose to first perform offline planning and then execute the plan, we propose here the OMPAS system, based on hierarchical operational models, which proposes to leverage planning as a guidance to an otherwise reactive system.

For these hierarchical operational models, this paper proposes both a Lisp-based language to define them, and its automatic analysis to generate predictive models of the system's behavior that can be leveraged by a hierarchical planner. The generation of predictive models allows to have a unified language for execution and planning, while having a rich structure to program operational models.

Introduction

In robotics, deliberation has always been a key component of autonomous systems. Endowing them with advanced deliberation capabilities allows them to be deployed in complex environments, in which they will be able to make wise decisions based on knowledge about the world collected via perception systems, the mission they have to carry out, and their own capabilities, while being able to cope with unexpected events.

Acting systems such as the *Procedural Reasoning System (PRS)* (Ingrand et al. 1996) and the *Refinement Acting Engine (RAE)* (Ghallab, Nau, and Traverso 2016) take advantage of a hierarchical representation of the agent skills, which allows reasoning about high-level goals, and the methods for achieving them by performing lower level tasks.

The agent skills can be represented as an acting domain A_Δ, a tuple (A, T, M_t) where A is the set actions (low-level skills), T the set of task (high-level skills), and M_t the set of methods to perform tasks in T, each method representing a different way to perform an abstract task $t \in T$. In acting systems, a method is an executable procedure that we call an operational model.

As skills of an agent are defined in A_Δ, the acting engine will use this knowledge to achieve an abstract task $t \in T$ by selecting a method m during the *task refinement* process. The refinement strategy of reactive systems is to execute an arbitrary method in a trial and repair manner until the mission succeeds. But such greedy approach has several shortcomings, as a suboptimal choice of method, can provoke inefficiencies, unnecessary failures or even lead the system into deadlocks. Such problems may be avoided by endowing the system with the capability of evaluating the long-term impact of a method. With such lookahead capabilities, the acting system is able to choose the method that will maximize its long-term success. One lookahead technique is planning, that exploits a model of the agent capabilities to modify its environment, to generate a sequence of commands called a *plan* to solve a goal.

As noted by Ghallab, Nau, and Traverso (2016), planning and acting require different models. Planners exploit a descriptive model of the environment that makes explicit the dynamics of the environment (the actor's "know what"). On the other hand, the acting part typically relies on operational models (the actor's "know how"), which are defined using general purpose languages, such that existing rich programming tools can be used to define the agent behavior (including, e.g., error handling, conditions, loops). This discrepancy in representations can be the source of many integration problems, in particular when the descriptive model diverges from the observed evolution of the environment and the agent's capabilities.

In this paper, we focus on an integrated planning and acting engine that is centered on a hierarchical operational model, from which a descriptive model can be extracted for planning purposes. We propose to exploit planning to make an informed choice when selecting a procedure or method for a given task. To do that we propose a new implementation of RAE named *Operational Model Planning and Acting System (OMPAS)*, together with a Lisp dialect based on the Scheme (Moretti 1979) variant to define operational models for RAE systems. Unlike previous implementations of RAE, this custom language enables the automated analysis of the operational model to extract planning domains from acting domains. We can then rely on existing planners (and in particular an extension of LCP (Bit-Monnot 2018) for hierarchical planning) to select the best method to achieve each task faced by the system.

Related Work

The deliberation community proposed several supervision approaches for acting systems. In the late 90s, PRS (In-

grand et al. 1996) was proposed as a goal oriented acting system. The skills are represented in a hierarchical fashion, which eases the definition of the problem and permits to divide the resolution into smaller and easier problems. Like in PRS, RAE (Ghallab, Nau, and Traverso 2016) needs to refine high-level objectives into a set of lower-level objectives. While PRS proposes a goal-oriented representation, RAE uses a hierarchical task representation closer to a Hierarchical Task Network (HTN) (Georgievski and Aiello 2014) representation, which eases the use of planning techniques to guide the refinement process.

To maximize long-term efficiency and success rate of the execution platform, many acting systems work jointly with automated planners. A first approach is to first call a planner, wait for a plan, and then execute it. *Propice-Plan* (Despouys and Ingrand 2000), proposed an architecture extending PRS with both *(i)* planning capabilities for long-term choices and *(ii)* an anytime anticipation module (based on simulated execution) which returns an estimation of the best method with an anytime algorithm. The addition of an anytime module makes the system more reactive, and less limited by the planner whose computation time largely dominates the one of other processes. *IxTeT-eXeC* (Lemai 2004) is yet another approach where acting and planning are interleaved, the planning process being continuously updated with task status and state updates to generate a valid plan.

Extensions to RAE have been proposed to endow the system with lookahead capabilities similar to *Propice-Plan* to guide the choice on the method to achieve the goal, taking into account both the current state of the system, and possible refinements of subtasks. RAEPlan (Patra et al. 2019) is an anytime planning system using MCTS techniques to sample results of low-level commands in each method, which gives two metrics to sort methods: their cost and efficiency. It has been extended with UPOM (Patra et al. 2020), taking into account the nondeterministic result of commands to guide the exploration of the search tree, and using learning techniques to speed up search and produce useful heuristics. Several techniques have been proposed along *UPOM* (i) Learnπ maps a tuple (τ, ξ) of a task and a state to a method, and is used when no time is allowed by the acting system to find a method, (ii) Learnπ_i returns the best instantiation of arbitrary parameters for a method in a given state ξ and LearnH gives heuristic for the branching in *UPOM*. While they do speed up search and increase the quality of the chosen methods, no guarantee is given on the long-term validity of the plan, and on the possibility to reach the high-level goal. In this paper we propose to address those issues by guiding the refinement process thanks to planning.

Recent works by Bansod et al. (2021) propose to extend the *Run-Lazy-Lookahead* algorithm first described by Ghallab, Nau, and Traverso (2016) with *Run-Lazy-Refineahead*. While the former executes a plan π as far as possible, calling Lookahead again only when π ends or a plan simulator says that π will no longer work properly, *Run-Lazy-Refineahead* proposes to extend the former one with a hierarchical representation of the domain. The algorithm can be linked to any online HTN planner that provides the solution as a refined task network, and that provides control over its backtracking features. This extension takes into account the whole refinement while checking the consistency of the plan, and thus fix some limitations of *Run-Lazy-Lookahead*. Bansod et al. (2021) put forward the integration of RAE with IPyHop, an HTN Planner extending the previous iteration of Python planners Pyhop and GTPyHop (Nau et al. 2021). The use of Python to define operational models is justified as an easier approach to define an acting domain, and makes it more accessible. However, it limits the automatic analysis of operational models, and the extraction of descriptive models. Therefore, the definition of the planning domain relies on dedicated descriptive languages such as PDDL (Fox and Long 2003) and ANML (Smith, Frank, and Cushing 2008).

System Overview

OMPAS is an acting system based on RAE. Skills of an agent are modelled as hierarchical operational models, where an operational model is defined as a procedure, encoded in a general purpose language, that can be executed by the acting engine. The hierarchical combination of operational models allows the definition of abstract tasks, that can be refined by several methods. We define an acting domain $A_\Delta\ (A, T, M_t)$ as the structure representing the skills of an agent, where A is the set of actions also called *primitive tasks*, achieving low-level goals, T the set of *abstract tasks*, achieving high-level goals, and M_t the set of *methods*, where each method $m \in M_t$ is a possible refinement of a task $t \in T$.

In this section, we present the overall architecture of *OMPAS* as well as the main refinement process. Finally, we propose a specific operational language based on the *Scheme* variant of *Lisp*: *Scheme OMPAS (SOMPAS)*.

Architecture of the system

OMPAS is a complete system that embeds the acting engine and a monitoring system, presented in Figure 1. The acting engine is connected to a *platform*, that is in charge of executing primitive commands and gathering the perceived state of the environment. In addition to update of the world, the platform communicates the current status of previously sent commands $\{$*Pending, Running, Failure, Success*$\}$. The monitor is an external process that can send tasks to the system or retrieve information about its internal status. The monitor can either be a program, or a human operator through the command line.

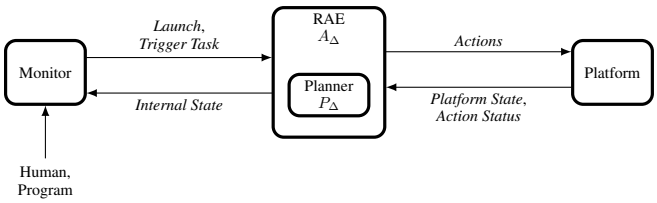


Figure 1: A global view on the architecture of *OMPAS*.

Refinement process

When facing a task $t \in T$ the role of the acting engine is to choose one applicable method $m \in M_t$ that will achieve the high-level objective of t. The system typically has the choice between several methods, each representing a particular procedure that might be used to achieve it. Initially, the refinement engine should *select* an arbitrary method $m \in M_t$ such that m is applicable in the current state, which is verified thanks to its *preconditions* that define the set of states in which the execution of m is possible. When a method m is chosen, its *body* is executed and should result either in a success (i.e. achievement of t) or a *failure*. The body of a method might contain arbitrary logic (if conditions, loops, ...), computations as well as the requirement to execute subtasks in T or actions in A. If m succeeds, the task t is considered successful. However, if m fails the refinement system resorts to a *retry* procedure where it selects another (not previously tried) applicable method $m' \in M_t$ and executes it. This procedure continues until either *(i)* a method succeeds, in which case the task is successful or *(ii)* all applicable methods have been tried unsuccessfully in which case the task is considered *failed*.

This task execution procedure is the same regardless of whether the execution was requested by an external process (t is a top level task) or as a result of executing a method of which t is a subtask.

Choices and programs In the previous definition of the refinement process, it should be obvious that the algorithm will have to make choices, primarily regarding the choice of the method to execute. Ideally we would like the first selected method to be successful as the contrary will typically result in the controlled agent acting in useless, counterproductive or even dangerous manner. As the choice of the method is arbitrary, the refinement engine is free to use any heuristic to guide his choice. In this paper, we are interested in guiding this choice by the analysis of the impact of a method on the state evolution. A key requirement for this is the ability to inspect the body of methods to predict the behavior they will induce in the controlled agent.

Acting language

As methods can be thought of as programs, their definitions are made through an acting language. In this work we propose to use an *expression* oriented language based on *Scheme*. An *expression* can be either an *atom* (boolean, number, symbol or procedure), or a *list* of expressions. When the first element of a list is a function symbol, it can be evaluated as the application of the function with the rest of the list as parameters. As the language is expression oriented, we consider body of methods as a single expression. The execution of such a language is based on the recursive evaluation of the root expression by an *Eval* function (Algorithm 1). *Eval* takes as input an expression and returns its evaluation, resulting from the recursive computation of all subexpressions. As everything is defined as a symbol, an evaluation environment *env* is used to bind symbols to values (themselves expressions).

Algorithm 1 Overview of the recursive evaluation of expressions in Lisp

```
function EVAL(expr, env)
    if expr is an atom then return VALUE(expr)
    else if expr is a list then
        f ← EVAL(expr[0], env)
        args ← []
        for i do in [1..|expr|]
            arg ← EVAL(expr[i], env)
            PUSH(args, arg)
        return APPLY(f, args)
```

The core of the implementation is based on the operators *define* (binds a symbol to an expression in env), *begin* (evaluates expressions sequentially and returns the result of the last one), *if* (evaluates expression conditionally), *quote* (returns the expression without evaluation) and *lambda* (creates a user-defined procedure). The language being purely functional, we consider all computations as *pure* (i.e. the result of a function application depends only on its parameters and has no side effects), except for a few identified acting primitives.

Acting primitives On top of Scheme, the acting language proposes several primitives offering ways to program parallelism, concurrency, and synchronization in the body of method. To keep the presentation focused, we only present here a restricted set of acting primitives that interest us in the present matter. Note that the following primitives are considered as impure.

- *exec-task* takes as argument a task label and a list of instantiated parameters. When it is an action, the engine sends a request to the platform to execute it. In the case of an abstract task, the refinement engine is called to find a suitable method to execute.
- *read-state* takes as argument a state-variable and returns its value at the time of evaluation.

Domain definition Several facilities are provided to define an acting domain using the same language. We illustrate them with the specification of the well-known *gripper* domain. A robot *robby* should transport balls in different rooms. Robby is equipped with two grippers *left* and *right*, each gripper can carry one *ball*. The domain contains 3 primitive actions: *(move ?from ?to)*, *(pick ?ball ?room ?gripper)* and *(drop ?ball ?room ?gripper)*. The abstract task *(pick-and-drop ?b ?r)* transports the ball *?b* to room *?r*. It can be refined by two methods *m-noop*, when it is already in the target room, and *m-pick-and-drop* for other cases. The state is represented via 3 state-functions: (i) *at-robby* : *room* returns the position of *robby*, (ii) *at* : *ball* $\rightarrow$ *room* returns the position of a ball or *no_place* if *robby* is carrying the ball, (iii) *carry* : *gripper* $\rightarrow$ *ball* returns the ball that a gripper is holding or *no_ball* if the gripper is empty. The following list of functions defines the acting domain.

- *def-state-function* takes as parameter the label of the state-function, a list of parameters with their annotated types, and a typed result.

```
(def-state-function carry
 '(?g gripper) '(?b ball))
```

- *def-action* takes as parameter the action label and a list of typed parameters.

```
(def-action drop '(?obj ball)
 '(?room room)
 '(?gripper gripper))
```

- *def-task* takes as parameter the task label and a list of typed parameters.

```
(def-task pick-and-drop
 '(?ball ball)
 '(?room room))
```

- *def-method* takes the following list of parameters:
  :task the label of the task the method refines.
  :params the list of typed parameters of the method. It contains the same list of parameters as the task, and more if necessary.
  :pre-conditions a list of expressions that returns *true* if a method is applicable, an error otherwise.
  :body the operational model of the method.

```
(def-method m1
'((:task pick-and-drop)
  (:params (?ball ball)
    (?room room)
    (?gripper gripper)
    (?departure room))
  (:pre-conditions
      (= (carry ?gripper) no_ball)
      (= ?departure (at-robby)))
  (:body
    (do
      (if (!= (at ?ball) (at-robby))
        (move ?departure (at ?ball)))
      (pick ?ball ?departure ?gripper)
      (move ?departure ?room)
      (drop ?ball ?room ?gripper)))))
```

- **Types and constants**: A domain can define new types and constants. As the engine does not support undefined values for state-variable yet, we define constants *no_ball* to represent the absence of ball, and *no_place* for the representation of the lack of information about the exact position of a ball.

```
(def-types room gripper ball)
(def-constants
     '(left right gripper)
     '(no_place room)
     '(no_ball ball))
```

In this section, we presented OMPAS, a supervision system using its own acting language to define operational models. The following section describes our exploitation of this language to extract descriptive models in the form of chronicles and the of a constraint-based planner to generate a plan.

Intermediate Representation of a Program

For our purpose of using planning as guidance of the refinement process in RAE, we propose to extract high-level descriptive models from the body of methods. The extraction process requires an intermediate representation that is more structured and amenable to automated analysis. In this section, we propose an interpretation of an expression (e.g. the body of a method) into a *Single Static Assignment (SSA)* form. A program in SSA has a structure like the following one:

$$\begin{aligned}&\dots\\&t_1 : r_1 \leftarrow \textit{prim-expr}_1\\&t_2 : r_2 \leftarrow \textit{prim-expr}_2\\&t_3 : r_3 \leftarrow \textit{prim-expr}_3\\&\dots\end{aligned}$$

where each t_i uniquely identifies a line, r_i is a label associated to the value computed by a primitive expression that is only allowed to refer to previous labels. This intermediate representation defines all steps in the evaluation of an expression in terms of the low-level capabilities of the interpreter.

To derive this representation, we start from a single line,

$$t : r \leftarrow \textit{body}$$

where $body$, a non-primitive expression, is the body of the method. Each line containing a non-primitive expression is systematically replaced by one or multiple lines based on a set of rules reflecting how each non-primitive expression should be interpreted.

For any SSA statement of the form,

$$t : r \leftarrow expr$$

we define below the rules for expanding the line based on the form of $expr$.

***expr* is an atom** that evaluates to the value v. The corresponding SSA representation is the following:

$$t : r \leftarrow \textit{cst}(v)$$

***expr* is a list** $(f\ e_1 ... e_n)$ In *SOMPAS* this corresponds to the application of the function f to the e_i parameters (where each e_i might be an arbitrary expression). Following the definition of the *Eval* function, it is expanded to:

$$\begin{aligned}&t_0 : r_0 \leftarrow f\\&t_1 : r_1 \leftarrow e_1\\&\dots\\&t_n : r_n \leftarrow e_n\\&t_{n+1} : r \leftarrow \textit{apply}(r_0, r_1, \dots, r_n)\end{aligned}$$

Note that after this expansion, other expansions will be triggered to, e.g., refine the computation e_1 into primitive expression or specialize the last line (function application) into a primitive expression depending on the nature of r_0.

***expr* matches** $(define\ var\ val)$ The operator defines a name for the value val and returns *nil*. It is translated as

$$t_1 : r_1 \leftarrow val$$
$$t_2 : r \leftarrow nil$$

and all subsequent uses of var name are replaced by r_1.

***expr* matches** $(begin\ e_1 ... e_n)$ The operator *begin* evaluates sequentially a list of expressions, and returns the result of the last expression. It is translated as

$$t_1 : r_1 \leftarrow e_1$$
$$\dots$$
$$t_n : r \leftarrow e_n$$

where the original label r is given the value of the last expression e_n.

***expr* matches** $apply(r_0, r_1, \dots, r_n)$ where r_0 is a **user defined function** f with parameters $x_1 \dots x_n$. In this case we replace the expression by the body of the function f where each parameter x_i has been substituted by the corresponding r_i value.

$$t : r \leftarrow body(f)[x_i/r_i]$$

***expr* matches** $apply(r_0, r_1, \dots, r_n)$ where r_0 is the *read-state* primitive:

$$t : r \leftarrow \textit{read-state}(r_1, \dots, r_n)$$

***expr* matches** $apply(r_0, r_1, \dots, r_n)$ where r_0 is a **task symbol**. In the acting engine, a task is either a primitive task, or an abstract task that is refined into a method, and the task ends when the method ends.

$$t : r \leftarrow \textit{exec-task}(r_0, r_1, \dots, r_n)$$

***expr* matches** $(if\ cond\ a\ b)$ The operator *if* first evaluates the expression *cond* that returns a boolean atom. If *cond* is true, *a* is evaluated and is the result of the expression, otherwise *b* will be evaluated. This is a form of conditional evaluation of the a and b expressions for which we rely on the existing semantics of tasks and methods. Hence, it is translated as, first an evaluation of the condition, and then the invocation of a newly created task τ that is given as parameter the result of *cond*.

$$t_1 : r_1 \leftarrow cond$$
$$t_2 : r \leftarrow \textit{exec-task}(\tau, r_1, \dots)$$

The synthetic task τ must be associated with two methods: one with precondition r_1 and body a and one with precondition $\neg r_1$ and body b. Note that for completeness, the synthetic task τ should also be given as parameter any variable that appears in the body of either a or b.

***expr* matches** $(quote\ e)$ The operator *quote* avoids evaluating an expression. The expression is therefore considered as an atom, and we obtain

$$t : r \leftarrow cst(e)$$

```
(do
 (move (at-robby) (at ?ball))
 (check (= (at-robby) (at ?ball)))
 (pick ?ball (at ?ball) ?gripper)))
```

(a) The body of a method that first *move* to the location of a ball, *check* if the action succeeded, and then *pick* the ball.

$$t_1 : r_1 \leftarrow \textit{cst}(move)$$
$$t_2 : r_2 \leftarrow \textit{cst}(\textit{at-robby})$$
$$t_3 : r_3 \leftarrow \textit{read-state}(r_2)$$
$$t_4 : r_4 \leftarrow \textit{cst}(at)$$
$$t_5 : r_5 \leftarrow \textit{cst}(value(?ball))$$
$$t_6 : r_6 \leftarrow \textit{read-state}(r_4, r_5)$$
$$t_7 : r_7 \leftarrow \textit{exec-task}(r_1, r_3, r_6)$$

(b) Part of the SSA form.

Figure 2: An example of conversion of the body of a method into SSA form.

Termination If no more rules are applicable, then the program has been transformed into SSA, meaning that all expressions are in primitive form. In our case it means that each statement corresponds to one of the primitives: *cst*, *read-state*, *exec-task* or *apply* (with the restriction that the first parameter of *apply* must be a built-in function as all user-defined ones have been expanded).

A partial example of the conversion from a method into SSA form is given in Figure 2.

Translation into a Planning Model

Given a method m with a body in SSA form, we want to translate it into a form that is amenable for planning using a *hierarchical temporal planner*. For this, we first discuss how causes of failures can be identified when constructing the SSA form and then propose a way to encode the method into *hierarchical chronicles*, a rich temporal model for primitive and abstract tasks (Godet and Bit-Monnot 2022).

Excluding error cases

The objective of planning is to find a choice of methods that would not result in any failure, i.e., that the final return value r is not *err* ($r \neq err$). This imposes a first requirement on the output of the program that can be propagated to other intermediate values.

Notably, an if-expression *(if cond a b)* is successful, if and only if the chosen branch is successful. If a branch cannot succeed then it must never be taken. For instance, if a is the *err* symbol, one can infer that $cond$ should always be false in order to lead to a successful execution. Two notable cases where this happens is in the *do* and *check* constructs that are syntactic sugar around if expression where one branch returns an error. If a $(do\ e_1 \dots e_n)$ is required to succeed, then all intermediate expressions are required to succeed as well. If a *(check c)* is required to succeed, then c is required to be true.

During the translation into SSA, this identification of necessarily successful outcomes and associated boolean constraints on conditions is done mechanically at each expansion, leading to a set of requirements on intermediate values.

Methods as chronicles

Having this SSA representation, it becomes quite natural to express them as predictive models adapted for planning and in particular into the formalism of *hierarchical chronicles* of Godet and Bit-Monnot (2022).

Each chronicle is associated to the task achieved by the method it represents. For a given method, a chronicle will contain a temporal variable for each of the t_i timepoints and a variable for each of the r_i intermediate results. In addition, a chronicle allows the definition of conditions that we exploit to represent the *read-state* primitive and subtasks that we exploit to represent the *exec-task* primitive. The other primitives will be translated as constraints, restricting the set of valid values for the variables.

For a given method, the corresponding chronicle is constructed by iterating through the SSA form and applying for each line the corresponding rule below.

Constant $t_i : r_i \leftarrow cst(v)$
The constraint $r_i = v$ is added to the chronicle.

Read state $t_i : r_i \leftarrow$ *read-state*$(sv, p_1, \dots, p_n)$
The condition $[t_i, t_i]\ sv(p_1, \dots, p_n) = r_i$ is added to the chronicle.

Subtask $t_i : r_i \leftarrow$ *exec-task*$(\tau, p_1, \dots, p_n)$
The subtask $[t_i, t'_i]\ \tau(p_1, \dots, p_n)$ is added to the chronicle, together with variable t'_i and the constraint $t_i \leq t'_i \leq t_{i+1}$. Note that in a hierarchical planner, all subtasks of a method must be successfully decomposed and do not provide any result value. The above translation is thus only valid when the r_i result was identified as necessarily successful by the error propagation.

Function application $t_i : r_i \leftarrow$ *apply*$(f, p_1, \dots, p_n)$
A new variable r_i is added to the chronicle, together with the constraint $r_i = f(p_1, \dots, p_n)$. Note that by construction, f is a built-in function so it is assumed that the solver is able to handle it, possibly with semantic attachment.

Success and ordering constraints We rely on the error analysis, to enforce that the plan is failure free if deemed valid by an automated planner. In particular, if a condition was detected as requiring a particular value, we enforce this as an additional constraint on the corresponding variable. We also enforce the order in the different statements, that is $t_i \leq t_{i+1}$ for any two subsequent timepoints in the SSA.

Post processing

With this automated process, we would like to obtain a chronicle such as the one of Figure 3. However, our automated conversion of a method into a chronicle has several shortcomings: redundant constraints can be present, variables might be duplicated and some timepoints might not be used in expressions of the chronicle. We propose several post-processing steps in order to simplify the resulting chronicle, and speed up planning by removing unnecessary variables and constraints.

$$
\begin{aligned}
variables &: \{s, e, ?ball, ?gripper, r_1, r_2, r_3, r_4, t_1, t_2\} \\
constraints &: \{r_1 = r_2, r_3 \neq r_4, \\
&\qquad s \leq t_1 \leq t_2 \leq e\} \\
conditions &: \{[s, s]\ at(?ball) = r_1 \\
&\qquad [s, s]\ at\text{-}robby() = r_2 \\
&\qquad [t_2, t_2]\ at\text{-}robby() = r_3 \\
&\qquad [t_2, t_2]\ at(?ball) = r_4\} \\
subtasks &: \{[s, t_1]\ move(r_2, r_1) \\
&\qquad [t_2, e]\ pick(?ball, r_4, ?gripper)\}
\end{aligned}
$$

Figure 3: The chronicle resulting from the conversion of the method of figure 2a (after post processing).

Variable binding The program analyzes the set of constraints, and for each equality constraint $(e_1 = e_2)$, such that e_1 and e_2 are atoms, binds the atoms following some rules depending on the *kind* of the atoms. Three kinds of atoms are distinguishable: (i) *constant* values, (ii) *parameter* variables of the method and (iii) *local* variables, which are generated during the conversion into the SSA form (as *timepoints* and *results*). We use an algorithm inspired by the *union-find* algorithm (Charguéraud and Pottier 2019) to bind two atoms a_1 and a_2, such that binding a_2 to a_1 means that *(i)* a_1 replaces a_2 everywhere it appears in the program and *(ii)* the allowed values of a_1 are restricted to only contain allowed values of a_2 (i.e. their domains are intersected).[1] Note that as a consequence of an unsatisfiable constraint, this domain restriction might result in a variable with no allowed value (which is also a contradiction). Given two atoms a_1 and a_2 required to be equal, we propose to bind a_2 to a_1 in the following cases:

- a_1 is a *constant* and a_2 a *variable* (parameter or local)
- a_1 is a *parameter* and a_2 a *local*
- a_1 and a_2 are of the same kind. Note that this last case is symmetric with respect to a_1 and a_2, so either a_1 or a_2 could be substituted.

After binding, we remove the constraint from the chronicle.

Simplification of temporal constraints During the conversion process, some unnecessary constraints with timepoints not involved in any expression of the chronicle are still in the set of constraints. We call those timepoints *ghost timepoints*, and propose an algorithm to check if such timepoints can be removed without modifying the properties of the chronicle. For this purpose we use Point Algebra (PA) as defined by Gerevini (2005), to first check path consistency of the *Temporal Network* resulting from constraints of the chronicle, and then shrink the set of constraints by removing *ghost timepoints*.

[1] To make this fully general, note that a constant can be interpreted as a variable with a single allowed value.

We define $p : (timepoint, timepoint) \rightarrow \{<, >, =, \leq, \geq, \neq, \top, \bot\}$ as the PA relation between two timepoints. Any *ghost timepoint* t' can be certainly removed from the chronicle in the following cases:

- t' has at most one relation. Any relation it is implied in can also be removed from the chronicle.
- t' has two relations with timepoints t_1 and t_2, with $p_1 = p(t_1, t')$, $p_2 = p(t', t_2)$, and $p_3 = p(t_1, t_2)$ the respective paths between t_1 and t', t' and t_2, and t_1 and t_2 in the network. We define $p'_3 = p_1 \circ p_2$ as the path between t_1 and t_2 through t' and state that p_3 can be replaced by $p''_3 = p_3 \cap p'_3$ the resulting path constrained by both p_3 and p'_3 as path consistency has been checked beforehand. Relations p_1 and p_2 are then removed from the chronicle, and p_3 is overwritten by p''_3.

Here is an example where t_2 and t_3 can be removed from the chronicle.

$$\begin{aligned} variables &: \{t_1, t_2, t_3, t_4, r_1\} && \rightarrow \{t_1, t_4, r_1\} \\ constraints &: \{t_1 \leq t_2 \leq t_3 < t_4, && \rightarrow \{t_1 < t_4\} \\ & \quad t_1 \leq t_4\} \\ subtasks &: \{[t_1, t_4](t\ r_1)\} && \rightarrow \{[t_1, t_4](t\ r_1)\} \end{aligned}$$

Merge conditions To avoid duplications of read-state of the same state-variables at the same instant, the program merges conditions c_1 and c_2, such that:

$$\begin{aligned} c_1 &: [s_1, e_1]\ sf(a_1, ..., a_n) = x \\ c_2 &: [s_2, e_2]\ sf(b_1, ..., b_n) = y \end{aligned}$$

If $s_1 = s_2 \wedge e_1 = e_2 \wedge a_1 = b_1 \wedge ... \wedge a_n = b_n$, then x and y are bound, and c_2 is removed. Here is an example of use case and the resulting chronicle.

$$\begin{aligned} variables &: \{s, ?b1, x, y, r1, r2\} && \rightarrow \{s, ?b1, x, y, r1\} \\ constraints &: \{r1 = x, r2 = y\} && \rightarrow \{r1 = x, r1 = y\} \\ conditions &: \{[s]\ at(?b1) = r1 && \rightarrow \{[s]\ at(?b1) = r1\} \\ & \quad [s]\ at(?b1) = r2\} \end{aligned}$$

Simplify constraints We propose to simplify constraints of the form $(true = c)$ where c is a constraint and replace it directly by c, which gives $(true = c) \rightarrow c$.

Remove useless variables During the translation into the SSA form, many temporal and result variables are created. The set of variables is analyzed to remove from the chronicle any local variable l which is neither in the set of *constraints*, *conditions*, *effects* nor *subtasks*.

Planning-based Method Selection

Planning-based lookahead while Acting

Having now a hierarchical predictive model of available methods, we would like to exploit it to guide the acting system in the selection of the method to apply when facing a task t to execute, using a hierarchical planner.

This is done through the PLANSELECTMETHOD of Algorithm 2. The most important part of the algorithm is the invocation of the planner (line 7) that, based on the current state ξ, will attempt to find a plan that refines the task t. This plan is then analyzed (line 8) to identify the method m that is used to refine t. If this succeeds and the method is one of the allowed methods in M, then the method is returned to be attempted by the acting engine (line 10). Otherwise, the system might resort to an arbitrary heuristic to select one of the allowed methods (line 12). The latter case might notably occur if the translation into chronicles fails (e.g. due to unsupported language features in the translation) or if the planner fails to find a plan within its allocated time.

Note that if t is a subtask of a method that was selected based on a plan π, then t will also appear in this earlier plan. When this occurs it is possible to check whether the previous plan π is still valid and if this is the case, avoid calling the planner to compute a new one (lines 5-6).

Algorithm 2 Plan-based method selection for a task τ.

```
1:  function PLANSELECTMETHOD(τ)
2:      m ← ∅
3:      ξ ← GET-STATE
4:      M ← APPLICABLE(τ, ξ) \ TRIED(τ)
5:      π ← GET-PARENT-PLAN(τ)
6:      if π = ∅ or not IS-VALID(π) then
7:          π ← CALL-PLANNER(τ, ξ)
8:      m ← GET-METHOD-FROM-PLAN(π, τ)
9:      if m ≠ ∅ ∧ m ∈ M then
10:         return m
11:     else
12:         return arbitrary m' ∈ M
```

Generation of the planning problem

In order to implement the CALLPLANNER function, we need a set of chronicles in order to represent *(i)* the initial state and objectives, *(ii)* the available actions and *(iii)* the available methods. While we have already covered how to generate a chronicle template for each method, we now turn our attention to the generation of the initial chronicle and of the action models.

Initial state generation The initial chronicle C_0 is instantiated with the current state ξ, such that for each state variable $sv(p_1, ..., p_n)$ with value v of the system, we define an initial effect $[0, 0]\ sv(p_1, ..., p_n) \leftarrow v$. The task t to be refined is defined as a subtask of the initial chronicle.

Assuming we want to select a method for a task *(pick-and-drop b1 kitchen)* in a state where *robby* is in the *living-room* and the ball is currently in the *bedroom*, we would obtain a state like the following

$$\begin{aligned} \textit{effects} &: \{[0, 0]\ \textit{at}(b1) \leftarrow \textit{bedroom}, \\ &\qquad [0, 0]\ \textit{at-robby}() \leftarrow \textit{living-room}, \dots\} \\ \textit{subtasks} &: \{[0, t]\ \textit{pick-and-drop}(b1, \textit{kitchen})\} \end{aligned}$$

In nominal functioning, OMPAS will take a snapshot of the state returned by the platform. We can also use OMPAS as a simulator, in which the execution is simulated by the evaluation of the operational models of actions.

Primitive Action Model To state the planning problem, we suppose in this work that models of actions are available and can be converted into chronicles. In practice, we exploit a PDDL-like (Fox and Long 2003) representation of an action, with *:params* a set of typed parameters, *:pre-conditions* a set of pre-conditions, *:effects* a set of effects.

```
(def-action-model move
'((:params (?from room) (?to room))
  (:pre-conditions
     (= (at-robby) ?from)
     (!= ?from ?to))
  (:effects
    (assert 'at-robby ?to))))
```

The above *def-action-model* macro expresses this as an operational model that can be converted into a chronicle (using the same procedure as for methods). Note the presence of the *assert* keyword that sets the value of a state variable. It is only allowed in primitive actions and will *(i)* be mapped to a new *set-state* primitive in SSA and *(ii)* results in an effect in the chronicle describing the action.

Preliminary Results

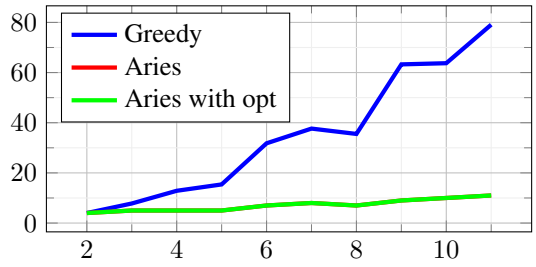
We experimented the present integration with Aries[2] (a hierarchical extension of the LCP planner) on a selection of domains. One of them is *gripper-door* in which several robots must displace balls in rooms that are connected by doors.

An abstract task *t_move* is used for the displacement of a robot in a target room, which associated methods are *m_noop* that is applicable if the robot is already at the goal, and *m_recursive* that is applicable if the robot is not at the goal position, the method performs the action *move* to an arbitrary location that is connected to the current position of the robot, and call again *t_move*. As in *gripper*, the state is defined by at, and $carry$; *at-rob* : $robot \rightarrow room$ gives the position of a robot, and $connected : (room, room) \rightarrow bool$ states if it is possible to pass from a room to another. Note that in some problems, some doors can be passed in one direction only, possibly leading robots into loops or dead ends.

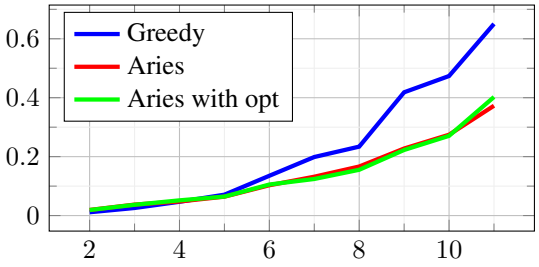
In this work we compare (1) the reactive system, (2) the system using Aries, and (3) the system using the optimal version of Aries (Aries-opt) in terms of total number of primitive actions and total refinement time to complete the task.

The plots in Figure 4 present the mean results on a series of runs for each of the 10 problems. For each problem we ran in simulation 30 times in reactive mode, 10 times with Aries, and once with Aries-opt. For small problems, both the total refinement time and the number of executed actions are equivalent between different techniques. However, as the complexity of the problem increases with the addition of rooms and connections, both Aries and Aries-opt outperform the purely reactive system on the total refinement time and the number of actions. This is explained by the structure of the problems, which makes the reactive system perform useless *move* actions, as it has no heuristic on the distance he is from the final destination, while planning systems finds a path in the topology of rooms. We can note that Aries and Aries-opt find very quickly solutions, with the planning cost compensated by the absence of failures (which in turn reduce the burden of selecting a recovery method).

[2]https://github.com/plaans/aries



(a) Mean number of executed actions in function of the number of rooms defined in the problem. Note that the number of actions is strictly the same using either Aries or Aries-opt.



(b) Mean of the total refinement time (in seconds) in function of the number of rooms defined in the problem.

Figure 4: Comparison of the number of executed actions and the total refinement time for several sizes of problems in the *gripper-door* domain in simulation.

Conclusion and Future works

The present work proposed a successful integration of a reactive system capable of generating temporal planning problems from the automatic analysis of operational models.

While a limited number of primitives are currently analyzed, our preliminary results on selected acting domains show promising results on the capacity of the system to produce a totally refined plan to advise the supervisor on the method to choose. Even if those results are on simple acting domains, the approach is general enough to adapt to larger and more complex acting domains, and will be tested in the continuity of this work.

We aim to extend our system to support further programming patterns, such as loops, and translate it as a planning representation, to tackle a wider set of problems, and enrich its capacity to use planning on more acting domains. One limitation of the current work is that the translation requires the total success of subtasks, and does not take into account the capability of an operational model to handle failures locally. Following iterations of the system will address this problem by taking into account the result of the execution of a task in the chronicle representation.

References

Bansod, Y.; Nau, D.; Patra, S.; and Roberts, M. 2021. Integrating Planning and Acting With a Re-Entrant HTN Planner 9.

Bit-Monnot, A. 2018. A Constraint-Based Encoding for Domain-Independent Temporal Planning. In Hooker, J., ed., *Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science, 30–46. Springer International Publishing.

Charguéraud, A.; and Pottier, F. 2019. Verifying the Correctness and Amortized Complexity of a Union-Find Implementation in Separation Logic with Time Credits. *Journal of Automated Reasoning* 62(3): 331–365.

Despouys, O.; and Ingrand, F. F. 2000. Propice-Plan: Toward a Unified Framework for Planning and Execution. In Goos, G.; Hartmanis, J.; van Leeuwen, J.; Biundo, S.; and Fox, M., eds., *Recent Advances in AI Planning*, volume 1809, 278–293. Springer Berlin Heidelberg.

Fox, M.; and Long, D. 2003. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *Journal of Artificial Intelligence Research* 20: 61–124.

Georgievski, I.; and Aiello, M. 2014. An overview of hierarchical task network planning. *arXiv preprint arXiv:1403.7426* .

Gerevini, A. 2005. Incremental Qualitative Temporal Reasoning: Algorithms for the Point Algebra and the ORD-Horn Class. *Artificial Intelligence* 166(1-2): 37–80.

Ghallab, M.; Nau, D.; and Traverso, P. 2016. *Automated Planning and Acting*. Cambridge University Press.

Godet, R.; and Bit-Monnot, A. 2022. Chronicles for Representing Hierarchical Planning Problems with Time. In *ICAPS Hierarchical Planning Workshop (HPlan)*.

Ingrand, F.; Chatila, R.; Alami, R.; and Robert, F. 1996. PRS: A High Level Supervision and Control Language for Autonomous Mobile Robots. In *Proceedings of IEEE International Conference on Robotics and Automation*, volume 1, 43–49 vol.1.

Lemai, S. 2004. *IxTeT-eXeC: planning, plan repair and execution control with time and resource management*. Ph.D. thesis, Institut National Polytechnique de Toulouse-INPT.

Moretti, G. 1979. The λ-scheme. *Computers & Fluids* 7(3): 191–205.

Nau, D.; Bansod, Y.; Patra, S.; Roberts, M.; and Li, R. 2021. GTPyhop: A hierarchical goal+ task planner implemented in Python. *HPlan 2021* 21.

Patra, S.; Ghallab, M.; Nau, D.; and Traverso, P. 2019. Acting and Planning Using Operational Models. *Proceedings of the AAAI Conference on Artificial Intelligence* 33(01): 7691–7698.

Patra, S.; Mason, J.; Kumar, A.; Ghallab, M.; Traverso, P.; and Nau, D. 2020. Integrating Acting, Planning, and Learning in Hierarchical Operational Models. *Proceedings of the International Conference on Automated Planning and Scheduling* 30: 478–487.

Smith, D. E.; Frank, J.; and Cushing, W. 2008. The ANML language. In *The ICAPS-08 Workshop on Knowledge Engineering for Planning and Scheduling (KEPS)*, volume 31.