Enhancing Operational Deliberation in a Refinement Acting Engine with Continuous Planning

Jérémy Turi, Arthur Bit-Monnot, Félix Ingrand

LAAS-CNRS, Université de Toulouse, INSA, CNRS, Toulouse, France
jeremy.turi@laas.fr, abitmonnot@laas.fr, felix@laas.fr

Abstract

Recent technological developments in robotics and artificial intelligence may enable the deployment of robots in many aspects of our lives. As the complexity of robotic platforms increases, deliberation algorithms need to be improved, in particular to handle an increasing number of agents, to manage complex goals and tasks, and to evolve in more open environments in which unforeseen events should be dealt with autonomously. Among the deliberation functionalities deployed to provide the best level of autonomy: planning, monitoring, learning, observing, we focus here on acting.

We present OMPAS, a refinement-based acting engine that executes high-level tasks by refining them into a set of lower-level tasks and commands. OMPAS uses a custom Lisp dialect (SOMPAS) to define the behavior of the robotic agent. SOMPAS provides primitives to handle concurrency and resources, and allows the synthesis of planning models, thanks to the restricted core language, and the explicit identification of acting decisions. The engine has been extended to deploy a continuous planning module, using the synthesized models, to look ahead and guide the decisions of the acting system in order to e.g., avoid deadlock, or optimize the completion of several parallel tasks. We provide an evaluation of the overall approach on the control of a fleet of robots in a simulated logistic platform.

1 Introduction

With recent technological advances in robotic and artificial intelligence, an increasing number of processes are automated and replaced by robotic agents. While initially, only repetitive and clearly identified tasks were automated, the research on deliberation algorithms allows the deployment of more general automated agents, that can adapt to various situations thanks to their skills. The agent deliberative power allows it to act autonomously at a certain level, which depends on its abilities to (i) perceive its environment and have a representation of the world in which it operates, (ii) act to change the said world, and (iii) reason on which set of actions to perform to fulfill its goals while taking into account operational constraints such as deadlines, exogenous events, etc. The present contribution focuses on the third element, i.e. the deliberation capabilities of the agent.

As stated by Ghallab, Nau, and Traverso (2016), automated deliberation can be separated in two parts: a *planning* system generating a plan composed of a set of actions to perform; an *acting* system in charge of supervising the execution of plans and adapting the behavior of the agent to the current state of the system.

Both the *planning* and *acting* subsystems need models to represent the capabilities of the agent as a set of actions. Planning systems conventionally use descriptive models, which define an action using pre-conditions and effects, respectively defining the states in which the action is applicable and the state transitions resulting from its execution. Acting systems rely on operational models that are executable programs used to perform actions. They can be defined with any executable language, allowing the use of generic programming constructs such as branchings and loops, making them generally more expressive than descriptive models, the latter using specific descriptive languages such as PDDL (Fox and Long 2003) and ANML (Dvořák et al. 2014). Because the acting and planning systems are based on different models, the interaction of the two systems may be more difficult, especially if there is a semantic mismatch between the descriptive model and the operational model.

We propose to use the acting system *Operational Model Planning and Acting System (OMPAS)* (Turi and Bit-Monnot 2022a), an enhancement of the *Refinement Acting Engine (RAE)* (Ghallab, Nau, and Traverso 2016), targeting multiple agents and concurrent activities. OMPAS extends RAE by adding the native support of concurrency in operational models, and managing the interleaving of parallel tasks thanks to a system handling the acquisition of shared resources. *OMPAS* uses a hierarchical representation of the agent skills, here defined with *SOMPAS*, a dedicated acting language based on Lisp (Steele 1990), from which planning models can be extracted (Turi and Bit-Monnot 2022b).

The present contributions adds an online planning module to OMPAS, used to guide runtime deliberation in the acting engine. To do this, we use a hierarchical temporal planner that indefinitely searches and optimizes a plan for all current missions, while taking into account updates of the state, and new missions. The generation of planning models is improved to support the automated analysis of more operational models, and in particular models requiring concurrency and resource access. We compare our work to a previous implementation of *OMPAS* with the factory simulator *Gobot Sim* presented by Turi and Bit-Monnot (2022b).

2 Related Work

The study of deliberation architectures have shown the importance of coupling a planning and a supervision system to make a robotic agent autonomous, and several approaches have been explored in the literature.

Language-based systems such as RAE (Ghallab, Nau, and Traverso 2016) and the Procedural Reasoning System (PRS) of Ingrand et al. (1996) are based on the iterative refinement of tasks into executable procedures, composed of commands or tasks. The refinement is done at runtime, which avoids having to generate the full command sequence before executing. While PRS is a goal-oriented system, RAE is task-oriented, using a formalism that ressembles Hierarchical Task Networks (HTN), and should ease the integration of HTN planners. Propice-Plan (Despouys and Ingrand 1999) extends PRS to improve its refinement process, both with an anticipation module and continuous planning, each one of them working at a different horizon. In the same way, the refinement process of RAE has been improved by using an anytime planner such as UPOM (Patra et al. 2021), using rollouts in a Monte Carlo Tree Search (MCTS) reinforced with learning. While UPOM provides a solution at any time, it does not guarantee a valid plan as opposed to the Run-Lazy-Refineahead algorithm (Bansod et al. 2021). As RAE may face problems in scaling up, Dec-RAE (Li, Patra, and Nau 2021) is a formalization of the decentralized version of the RAE algorithms. OMPAS (Turi and Bit-Monnot 2022b,a) is yet another version of RAE that proposes a unified framework for planning and acting, using the dedicated acting language *SOMPAS* to define operational models, that support concurrency and resource sharing, and from which planning models can be extracted. However, previous versions of the automated generation of planning models lacked the analysis of concurrent and resource primitives, and planning was only used to guide local choices, which we generalize in the present work by using an online planner that takes into account all current tasks.

The *Plan-Exec* architecture is another approach to deliberation. Remote Agent (RAX-PS) (Muscettola et al. 1998) is the first successful integration of a deliberation architecture composed of a planner and an execution module on a space explorer. IDEA (Muscettola et al. 2002) and T-REX (McGann et al. 2007) propose a first attempt at a unified representation for the different level of abstraction of the system. They relied on constraint based languages which can make the programming of the agent challenging. Other systems such as IxTeT-Exec (Ingrand et al. 2007) and more recently FAPE (Bit-Monnot et al. 2020) use a Plan-Exec architecture based on temporal planners to take into account time, and dedicated execution modules, mostly used to monitor the execution of a fully instantiated plan. The CASPER/ASPEN system (Chien et al. 2000; Rabideau et al. 2000) integrates (i) a planner which plans at different levels of abstraction and horizons to improve the reactivity of the system, and (ii) iterative repair techniques and conflict checking systems making it more robust to hazard. ROSPlan (Cashmore 2015) and PlanSys2 (Martin et al. 2021) are two propositions of generic *plan-exec* frameworks to use planning in robotic systems; Both use planning models based on PDDL (Fox and Long 2003), and decouple planning from execution.

CRAM (Beetz, Mösenlechner, and Tenorth 2010) is another system using a Lisp dialect to program the agent behavior, in which parameters of skills can be instantiated at the very last moment to adapt more easily to runtime constraints, but does not propose a tight integration with a planning system to guide its choices.

Some works on Belief Decision Intention (BDI) Systems propose to optimize the overall behavior of the system by improving the interleaving of tasks. *Summary information* is used by Clement and Durfee (1999) to generate deadlock-free interleaving of plans. An interleaving of plans at the command level is proposed by Yao and Logan (2016) thanks to MCTS rollouts, improving performances compared a simple Round Robin, but does not ensure deadlock-free interleaving, which can be solved using sound planning. The system of Sardina, de Silva, and Padgham (2006) resorts to planning for some explicit choices defined in skills, which our work extends to any choice of the acting system.

3 Acting system

3.1 OMPAS, an extended version of RAE

OMPAS (Turi and Bit-Monnot 2022b) is an acting system that executes tasks by refining them down to a set of lower-level tasks and commands considering a model of the behavior of the agent, and the state of the system. The agent is modeled as a tuple (C, T, M) where C is the set of commands representing the low-level capabilities of the agent, T is the set of tasks corresponding to the high-level capabilities of the system, and M the set of methods corresponding to the skills of the agent. A method is an operational model that achieves a high-level task using lower-level tasks and commands. A method $m \in M$ is associated with a particular task $t \in T$, and is defined by a list of parameters (possibly inherited from t), pre-conditions that define the set of states in which the method is applicable, and a body defined in an executable language. The acting system supports a partially observable state and stochastics command outcomes. However, it has not been designed to support perception uncertainty, but no architectural limitations prevents it to support it in a future version. The project and its documentation are available at https://github.com/plaans/ompas

OMPAS uses a dedicated acting language to program the agent behavior using the Lisp dialect *SOMPAS*. The language has dedicated acting primitives to execute a command or a task to be refined with *exec*, get the current value of a state variable with *read-state* or select an arbitrary element from a set with *arbitrary*. The system has been extended by Turi and Bit-Monnot (2022a) to add the native support for concurrency in its main algorithm, and to expand the acting features of the system, by not only refining tasks into methods, but also managing resource requests in a dynamic way to improve the interleaving of tasks. Thus, *SOMPAS* uses concurrency primitives to start an execution in a new thread using *async*, to *await* its result, or *interrupt* the computation if necessary. A system of resources has been added to synchronize the execution of programs in *SOMPAS*. Resources can be manipulated using *acquire* to request its usage, and *release* to give the resource back. An example of an operational model using *SOMPAS* is presented in Figure 1 and features some new acting primitives.

```
params: ?p package, ?m machine
pre-conditions: none
body: (begin
       (define ?r (arbitrary (read-state
          instances robot)))
       (define h1 (acquire ?m))
       (define h2 (acquire ?r))
       (exec carry ?r ?p ?m)
       (release h2)
       (exec process ?m ?p))
```

Figure 1: An example of an operational model defined with *SOMPAS*. This model is a skill used in *Gobot Sim* that processes a package *?p* on a given machine *?m*. An arbitrary robot *?r* is selected, and as other tasks might be using *?r* and *?m*, both resources are acquired before using *?r* to bring the package *?p* to *?m*. Then, *?p* is processed on *?m*.

3.2 Guiding acting processes

In the previous versions of *OMPAS*, acting processes are locally and reactively handled, and do not take into account either the execution of other tasks, or possible future states of the system. To address both matters, we propose to use a planner in a continuous fashion to adapt to the evolution of the system, and guide all acting processes, which are: the refinement of tasks at runtime; the instantiation on the fly of arbitrary variables and; the allocation of resources to ongoing methods. The analysis of the returned plan should give, (i) the most promising method for each task not yet refined, (ii) instantiation for arbitrary variables, (iii) an order access to resources shared between current tasks.

3.3 Update of the deliberation architecture

Continuous planning requires the synchronization between the supervision system and the automated planning system. In order to do that, two new components are integrated to *OMPAS*. The first one is the planner that continuously looks for either a suitable solution to the planning problem of the current execution state, or a better solution in terms of global makespan (time to execute all the ongoing tasks). The second one is called the *Acting Manager* and keeps track of all acting processes that *OMPAS* faces. It both integrates the traces of executed acting processes, and those anticipated by the planner. By representing all acting processes in the *Acting Manager*, we simplify the update of the planner using traces of all executed acting processes, and give a unique interface to provide guidance for upcoming acting processes, simplifying the usage of a global planner. Figure 2 presents the interaction of the *Acting Manager* with, on one side, the *Execution Manager*, and on the other side, the *Planner*.

Acting and Execution Manager The *Execution Manager* is responsible for the supervision of the execution of all current tasks in separated threads. When an acting process a_p is executed, the *Execution Manager* requests guidance to the

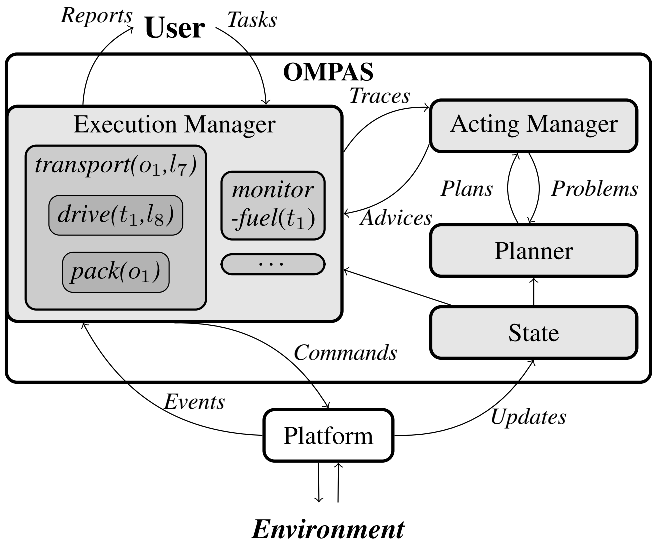

Figure 2: The new internal architecture of *OMPAS* that integrates the *Acting Manager* and the *Planner*.

Acting Manager. If a_p has been anticipated by the planner, the *Acting Manager* advises the supervisor of the choices the planner made. If the choices are still valid in the current context, the supervisor resorts to them to solve a_p. Otherwise, reactive algorithms are used to find a valid solution. For example if a task must be refined, and the planner found that the best to try method is m, the execution system will first check that m is still applicable in the current context, otherwise it selects an arbitrary method among those applicable. In all cases, the *Execution Manager* sends back the choices it made at runtime as well as other values such as start and end times of executed actions, which inform the process values and therefore constrain the planning problem.

Acting Manager and Planner As the planning problem should anticipate the execution state of the system, the *Acting Manager* updates the planner upon evolution of the system. Updates are triggered on the instantiation of parameters or timepoints, new tasks to address and updates of the state. The planner should take into account those updates, and either repair its plan, or replan if necessary. When the planner finds a new solution, it returns an instantiated plan from which acting choices can be extracted and stored in the *Acting Manager*.

Unified representation of acting processes The *Acting Manager* uses a unified representation of the state of the system to bind the execution traces to the corresponding variables in the planning model. The state is represented as an *Acting Tree*, where a leaf is an *Acting Process*, and represents the hierarchical links between processes. An example of *Acting Tree* is given in Figure 3. Any acting process is defined by an execution interval, and other variables depending on the process kind, which are the following:

- *Root*: virtual task that contains references to all the high-

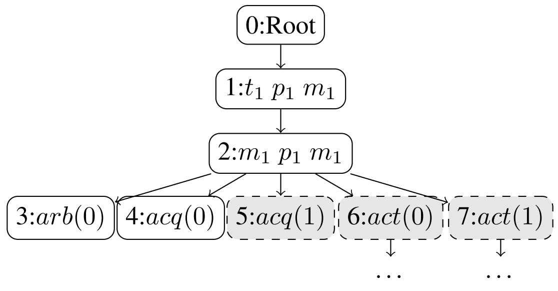

Figure 3: An example of *Acting Tree* of a task refined into the operational model defined Figure 1. The operational model is partially executed. The three remaining dashed and grayed processes have been planned, and will be used to guide the execution of the operational model when needed, but might adapt their values regarding evolutions of the system.

level tasks that *OMPAS* should address.

- *Action*: represents the execution of either a *command* or a *task*, and contains the list of arguments of the action, i.e. the label of the action and the value of its parameters. In case it is a *task*, it contains references to all the refinements of the task: current, failures and suggested.
- *Refinement*: method that refine a given task. It is defined by a list of arguments and a set of references to acting processes called within the method.
- *Arbitrary*: call to the acting primitive *arbitrary*. It contains the selected value.
- *Acquire*: represents the acquisition of a resource. It contains the name of the resource, the required quantity as well as the dates of request, acquisition and release of the resource.

Variables used in acting processes have two different kinds of value to differentiate instantiation by the *Planner* or the *Execution Manager*. When the planner finds an instantiation i for a variable v, the value is marked as *Planned* which means that i can be suggested to the *Execution Manager* upon request for guidance. The variable receives the *Executed* value e when the variable is updated by the *Execution Manager*, which transforms the planning variable as a constant with value e. This allows to have a unique variable used by both the planner and the execution, abstracting updates from both components.

Identification of processes As different components of the system must refer to the same *Acting processes*, we use a referring system to uniquely identify those processes. Processes are identified in two ways: a unique absolute *Id* set at runtime, and a relative *Label* defined by the kind of the process (Root, Action, Refinement, Arbitrary, Acquire) and an instance number unique in the context of the method. For example in Figure 3 we could either refer to the first *arbitrary* process using its *Id* 3, or relatively to the root process with the absolute path Root/Action(0)/Refinement(0)/Arbitrary(0), or the path relative to the method 2/Arbitrary(0).

4 Instantiation of the planning problem

In the precedent section, the OMPAS/SOMPAS system has been presented along its extension including a continuous planning module and an *Acting Manager* necessary to synchronize the execution and the planner. In this section, we present the selected planner, and how the planning problems are encoded using the *Acting Tree* of the *Acting Manager*.

4.1 Hierarchical temporal planner

As OMPAS uses hierarchical operational models, it seems more natural to look for a hierarchical planner. As the planner should guide the order access of resources, it should explicitly reason about time. The planner that seems to best meet our requirements is Aries, the extension of the *LCP* (Godet and Bit-Monnot 2022) for hierarchical planning. It features a representation of a planning problem using hierarchical chronicles that are defined by a tuple (V, X, C, E, S) where V is the set of variables (parameters) of the chronicle, X a set of constraints over V, C a set of timed conditions, E a set of timed effects, and S a set of subtasks.

4.2 Encoding of the problem

To use this planner, we need to define the planning problem P_Π as a tuple (C_0, C_I, P_Δ). C_0 is the initial chronicle that holds the known values of the state represented as a set of effects, including the initial state and the known transitions, as well as the goals and tasks to achieve, respectively encoded as conditions and subtasks in C_0. C_I is the set of chronicle instances present in the encoding of the problem, and P_Δ a set of chronicle template that can be used to refine subtasks of $C_0 \cup C_I$ that are not yet refined by any chronicle of C_I. Going back to the *Acting Manager*, the chronicle C_0 is constructed using both the *Root* process that contains all the high-level tasks, and the perceived states of the system. C_I is the set of *Refinements* of all current tasks that have been declared in the *Acting Manager*. P_Δ contains the templates for all actions and methods that the system may execute. Each template can be instantiated to refine a particular subtask of $C_0 \cup C_I$, in particular when a model is needed for an action not yet present in the *Acting Tree*.

4.3 Exploitation of the plan

With such encoding of the planning problem, we expect the planner to deliver useful information to guide the different acting processes which are: the refinement of abstract tasks, the selection of arbitrary values, and access order for the shared resources. As the returned plan is a set of instantiated chronicles, the plan is therefore analyzed in order to extract the preferred values for each Acting Variable. To do that, we define a set of *Acting Process Binding* that link the acting variables of an acting process to the corresponding variables in the chronicle. An *Acting Process Binding* uses the same reference system as the *Acting Tree* and the operational models, such that the sole instantiation of those bindings are needed to extract preferences from the plan.

Guidance of the resource acquisition One particular feature of OMPAS is its advanced management of resource allocation, allowing an external tool to reshape the waiting queue of each resource in order to optimize the overall process. Here we use the planner to give an access order for each resource. Using only the acquisition dates of each acquisition process of a particular resource, the *Acting Manager* classifies them in an increasing order, and requests acquisition of the resource with the appropriate priorities to respect the order given by the planner. However, we still want to support reactive emergency requests, such as taking control of a robot to recharge it. Therefore, we use a priority system to accept both reactive and planned acquisitions. We define a priority as a tuple *(strong, weak)*, where priorities are first sorted by their *strong* components in decreasing order, and then by their *weak* components in an increasing way. Reactive acquisitions are only using the *strong* part, whereas planned acquisition have a constant *strong* priority, and a variable *weak* priority.

To illustrate this system let us set the *strong* priority of planned acquisitions to 10. We have four requests to access a resource with the following priorities: {*(10,1), (1000, 0), (5,0), (10,2)*}. Once sorted the access order will be {*(1000,0), (10,1), (10,2), (5,0)*}. By having this double level of priority, we can sort requests from both the *Execution Manager* and the *Planner*.

To conclude, by using the planner continuously the *Acting Manager* can associate preferred values to the *Execution Manager* when requested by looking for the best instantiation for every *Acting Variable* that are still unbounded by the execution. The planner can also identify preferred refinement for tasks, which are suggested to the *Acting Manager* and used during the online refinement processes.

5 Conversion of operational models

The precedent sections showed that an architecture with an *Acting Manager* facilitates the use of any planner in an online fashion to guide acting. A specific planner has been selected that requests an encoding of the problem as chronicles. In the present section we will describe the techniques and rules used to extract chronicles automatically by analyzing the body of operational models defined in *SOMPAS*.

5.1 Two-phase conversion

During the early development of *OMPAS*, a first set of methods and rules have been presented to extract chronicles from the body of a method (Turi and Bit-Monnot 2022b). It was based on a two-phase conversion process. First the body was translated into a lower level formalism similar to *Single Static Assignment (SSA)*, which consists in translating each expression into a sequence of primitive instructions. Secondly, the SSA form was converted into a chronicle by translating each primitive expression into a set of constraints, conditions, effects and subtasks.

This two-phase conversion presented some advantages: by splitting the conversion into two set of rules, we can abstract the use of a specific language to define operational models. However, the SSA form presents some limits, in particular when it comes to represent concurrency. To address this matter, we propose to evolve the SSA formalism to a Flow Graph formalism to represent richer models.

Flow Graph Formalism A Flow Graph is a graph representation of all paths that might be traversed through a program during its execution. This representation is more suitable to check the coverage of a program, and the paths that lead to failures, deadlocks and dead-ends. Previous work focused on Control Flow Graph for Scheme-like language by Shivers (1988), however as SOMPAS is simpler than the generic Scheme, we have derived our own set of rules which are sufficient for the extraction of planning models. In this Flow Graph representation, we define a node of the graph as a flow, which is defined by an interval composed of two timepoints, a result, and a sequence of primitive instructions, which are similar to the ones in the SSA form. As in *SSA*, each label is uniquely defined, and the computation of each primitive expression depends only on previously defined labels.

5.2 Translation into a Flow Graph

The translation procedure follows the same steps as defined by Turi and Bit-Monnot (2022b): the flow graph is derived from a program by iteratively translating any expression into a set of flows and branches, until each flow is composed uniquely of a sequence of primitive computation. We start with a *body* expression, representing the program of the method we want to translate, which corresponding *flow graph* is composed of a unique flow as follows:

$$\boxed{[t]\ r \leftarrow body}$$

The Flow Graph representation of the *body* is obtained using a procedure based on the rules defined below. The procedure translates every flow of the form:

$$\boxed{[t]\ r \leftarrow expr}$$

A rule will be applicable if the *expr* expression has the required form. When applicable, a rule will translate a single flow node into a set of simpler flow nodes. Note that the first seven rules are adapted from the corresponding rules in SSA form (Turi and Bit-Monnot 2022b).

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

$$\boxed{[t]\ r \leftarrow cst(v)}$$

***expr* is a list** $(f\ e_0 ... e_n)$ 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:

$$\begin{tikzcd}
\boxed{[t_0, t_1]\ r_0 \leftarrow f} \arrow[d] \\
\boxed{[t_2, t_3]\ r_1 \leftarrow e_1} \arrow[d] \\
\cdots \arrow[d] \\
\boxed{[t_{2n}, t_{2n+1}]\ r_n \leftarrow e_n} \arrow[d] \\
\boxed{[t_{2n+2}, t_{2n+3}]\ r_{n+1} \leftarrow apply(r_0, r_1, \ldots, r_n)}
\end{tikzcd}$$

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

$$\boxed{\begin{array}{l}[t_0, t_1]\ r_0\ \leftarrow val \\ [t_2]\ r_1\ \leftarrow nil\end{array}}$$

***expr* matches** $(begin\ e_0 ... e_n)$ The operator *begin* evaluates sequentially a list of expressions, and returns the result of the last expression. It is translated as:

$$\begin{tikzcd}[row sep=small] \boxed{[t_0, t_1]\ r_1\ \leftarrow e_1} \arrow[d] \\ \dots \arrow[d] \\ \boxed{[t_{2n}, t_{2n+1}]\ r\ \leftarrow e_n} \end{tikzcd}$$

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.

$$\boxed{[t]\ r \leftarrow body(f)[x_i/r_i]}$$

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

$$\boxed{[t]\ r \leftarrow read-state(r_1, \dots, r_n)}$$

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

$$\boxed{[t]\ r \leftarrow exec(r_0, \dots, r_n)}$$

***expr* matches** $(if\ cond\ a\ b)$ The operator *if* first evaluates the expression *cond* that returns the boolean r_0. If r_0 is true, *a* is evaluated and is the result of the expression, otherwise *b* will be evaluated. The result of $expr$ is either the result of a or b, which depends on r_0.

$$\begin{tikzcd} & \boxed{[t_0, t_1]\ r_0\ \leftarrow cond} \arrow[dl, "r_0"'] \arrow[dr, "\neg r_0"] & \\ \boxed{[t_2, t_3]\ r_1\ \leftarrow a} \arrow[d] & & \boxed{[t_4, t_5]\ r_2\ \leftarrow b} \arrow[d] \\ \boxed{[t_6]\ r_{if}\ \leftarrow r_1} \arrow[dr] & & \boxed{[t_6]\ r_{if}\ \leftarrow r_2} \arrow[dl] \\ & \boxed{[t_7]\ r \leftarrow r_{if}} & \end{tikzcd}$$

***expr* matches** $(async\ e)$ The operator *async* evaluates in a new thread the expression e, and returns a handle used to refer to the concurrent execution, either to await on its result, or interrupt it. Here the handle is bound to the flow of the concurrent evaluation, and can refer to the execution interval of e and its result. It translates as two flows representing two threads bound by the timepoint t_0.

$$\boxed{[t_0]\ r_0\ \leftarrow h} \qquad \boxed{[t_0, h.end]\ h.result\ \leftarrow e}$$

***expr* matches** $(await\ h)$ The operator *await* holds the evaluation of the current thread until the result of the concurrent evaluation c_e is available, returning the result of c_e. The corresponding flow graph is:

$$\boxed{\begin{array}{l}[t_0]\ r_0\ \leftarrow h \\ [t_2, t_3]\ r\ \leftarrow await(r_0)\end{array}}$$

***expr* matches** $(arbitary\ r_1)$ In SOMPAS, the arbitrary primitive returns an arbitrary element of a set. The corresponding flow is:

$$\boxed{[t]\ r \leftarrow arbitrary(r_1)}$$

where r_1 should represent a finite set of elements, either a list of atoms, or a symbol type.

***expr* matches** $(acquire\ ?r\ ?q)$ The operator *acquire* requests a resource $?r$ with quantity $?q$, and holds the execution until the resource is granted. It returns a *resource-handle* that can be used to release the resource. The corresponding flow graph is:

$$\boxed{\begin{array}{l}[t_0]\ r_0\ \leftarrow ?r \\ [t_1]\ r_1\ \leftarrow ?q \\ [t]\ r \leftarrow acquire(r_0, r_1)\end{array}}$$

***expr* matches** $(release\ r_h)$ The operator *release* returns the borrowed quantity of the resource. The corresponding flow graph is:

$$\boxed{\begin{array}{l}[t_0]\ r_0\ \leftarrow r_h \\ [t]\ r\ \leftarrow release(r_0)\end{array}}$$

Static analysis of Flow Graph Once a program has been translated into its Flow Graph equivalent, static analysis is performed to simplify the model and detect invalid branches. Those post-processings are comparable to classical static passes in compilers such as type analysis, constant propagation, and pre-evaluation of some primitive expression, made possible by the side-effect-free semantics of most primitives of *SOMPAS*.

Specialization for Planning The analysis produces a computational model that matches the semantics of the operational model. Our objective however is more specific as we want to guide the acting system into an error-free execution path. Failures are represented as errors in the result of the computation of an operational model. We enforce that the result of an operational model cannot be of type *Err*, and therefore remove the type error from its possible values. Then, using the typing system of SOMPAS and a type propagation mechanism, the system identifies flows that necessarily lead to errors, by looking for variables that have an empty type, meaning that the execution in those flows are invalid. Then, the system constraint choices that can be made to forbid taking those invalid path.

$$\boxed{\begin{array}{l}[s]\ arb \leftarrow arbitrary(\{robot1, robot2\})\\ [s, t1]\ r1 \leftarrow acquire(?m)\\ [t1, t2]\ r2 \leftarrow acquire(arb)\\ [t3, t4]\ nil \leftarrow exec(carry, arb, ?p, ?m)\\ [t4]\ r4 \leftarrow release(r2)\\ [t5, e]\ nil \leftarrow exec(process, ?m, ?p)\end{array}}$$

Figure 4: Translation of the operational model presented Figure 1 into its simplified Flow Graph representation. The resulting graph has been post processed to minimize the number of nodes and labels used to represent the program.

The Flow Graph in Figure 4 is an example of translation of the program defined in Figure 1.

5.3 Conversion of the primitives expressions to build the chronicle

After generating and analyzing the flow graph, a set of rules are used to build the corresponding chronicle, defined by a set of variables, a collection of timed conditions and effects parameterized with variables, and a set of partially ordered subtasks. We extend the set of already defined rules by Turi and Bit-Monnot (2022b) to support the conversion of the operators async, await, arbitrary, acquire and release.

Branching flows Branching flow of the form presented in the conditional (*if*) rule is converted as a synthetic task τ associated to two methods: the first one represents the left branch, and is composed of the pre-condition r_0 and the translation of the flows of the left branch, and the contrary for the second method representing the right branch.

$[t]\ r\ \leftarrow\ h$ **where h is a handle** A set of constraints arise from this primitive expression. First as h represents the spawn of a new thread, the constraint $h.start\ =\ t$ states that the evaluation starts the moment the thread is created. The semantic of the language states that if all references to h are released, then the asynchronous process of h is interrupted, which can be considered as a failure in the planning semantic. We define $Drops = \bigcup drop(r) : r = h$ the set of all moment the reference is dropped and enforce $h.end \leq max(Drops)$.

$[t_1, t_2]\ r\ \leftarrow\ await(h)$ As we await the process, we can add the temporal constraint $h.end \leq t_2$.

$[t]r \leftarrow arbitrary(set)$ An arbitrary flow is converted as a disjunctive constraint, such that for each element $e_i \in set$: $\bigvee r = e_i$

$[t_1, t_2]\ r\ \leftarrow\ acquire(r_0, r_1)$ To represent the acquisition of r_0 with quantity r_1, we use virtual state variables representing both the maximum and current capacity of a resource. We represent them as integers; atomic resources have a capacity of one. An acquisition is defined by adding

$$\begin{aligned} variables &= \{s, e, p, arb, t1, t2, t3, t4, t5, t6, t7, t8,\\ &\quad r6, r7, r8, r9, r10, \dots\}\\ constraints &= \{s \leq t6, t1 \leq t2, t2 \leq t3, t4 \leq t5, t5 \leq e,\\ &\quad r6 = r9 - r10, r7 = r8 + r10, \dots\}\\ conditions &= \{[t6]\ capacity(?m) = r9,\\ &\quad [e]\ capacity(?m) = r8,\\ &\quad [s]\ max - c(?m) = r10, \dots\}\\ effects &= \{[t6, t1]\ capacity(?m) \leftarrow r6,\\ &\quad [e, t7]\ capacity(?m) \leftarrow r7, \dots\}\\ substasks &= \{[t3, t4]\ nil \leftarrow (carry\ arb\ ?p\ ?m)\\ &\quad [t5, e]\ nil \leftarrow (process\ ?m\ ?p)\}\end{aligned}$$

Figure 5: Partial chronicle resulting from the conversion of the flow graph of Figure 4. Some elements have been omitted to simplify the example.

the following elements to the chronicle

$$\begin{aligned} X &= \{q_1 = q_0 - q, q_3 = q_2 + q\}\\ C &= \{c1 : [t_2]capacity(r_0) = q_0\\ &\quad c2 : [t_3]capacity(r_0) = q_2\}\\ E &= \{e1 :]t_2]capacity(r_0) \leftarrow q_1\\ &\quad e2 :]t_3]capacity(r_0) \leftarrow q_3\}\end{aligned}$$

The value q is either r_1, or the constant $max - c(r_0)$ if r_1 is not present. The condition c_1 and effect e_1 represent the acquisition of the resource, and c_2 and e_1 the release and t_3 represent the release instant.

$[t]\ r\ \leftarrow\ release(r_h)$ The explicit release of a resource can be triggered with the *release* primitive and constrains the release date $acq_{end}\ =\ t$. If r_h can be released at various points in the program, we extend the constraint to $acq_{end}\ =\ min(Releases)$, where $Releases$ is the set of all release dates. It should be noted that if a resource is not released explictly, then $acq_{end}\ =\ max(Drops)$ where $Drops = \bigcup drop(r) : r = r_h$.

All those rules are used to build the chronicle of Figure 5 obtained from the Flow Graph of Figure 4. Once the chronicle is constructed, the same post-processings as described by Turi and Bit-Monnot (2022b) are used to simplify the structure of the resulting chronicle by removing unnecessary variables, and reduce the number of constraints, conditions, effects and subtasks.

6 Preliminary results

A first incomplete implementation of the presented system has been used to compare it to the previous version of *OMPAS*. It can produce a plan to guide all acting processes described above but could only be used in a Plan-Exec fashion, i.e., first producing a plan and then executing. It requires ad-

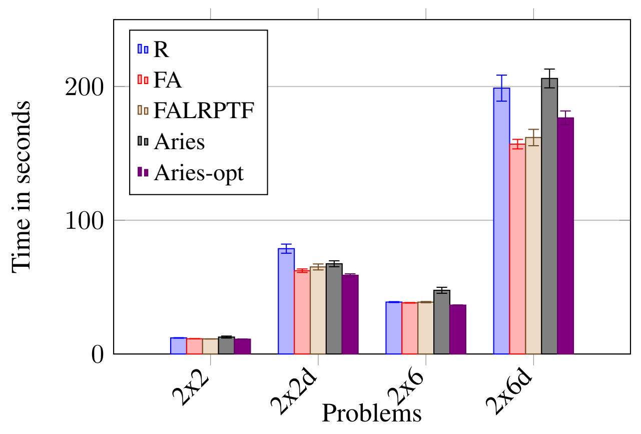

Figure 6: Comparison of the mean execution time on 10 runs to complete a task of $n * m(d)$, where n is the number of package and m the number of jobs, d means robot displacement are simulated. R, FA and FALRPTF are reactive strategies whereas Aries and Aries-opt resort to a planner to guide acting processes.

ditional engineering work to update the planner with execution traces and use it in a continuous fashion. We tested the system with a single high-level task: on reception of the mission, the system holds its acting deliberation until the planner produces a valid plan containing the full execution path. Then, the execution resumes, and this only plan is used to guide acting decisions, and in case the execution shifts from the plan, for example when a selected method to refine a task is no more applicable in the current context, OMPAS resorts to its reactive algorithms to act. The planner can be set up to either find a first suitable plan or return an optimal plan in terms of temporal makespan.

We tested the system on Gobot Sim (*https://github.com/plaans/gobot-sim*), a factory simulator , in which a fleet of robots is used to bring packages to different machines, on which specific processes can be done (Turi and Bit-Monnot 2022a). The goal is to treat all incoming packages, meaning doing all needed processes on a package, and then deliver it. We compared the system to the reactive strategies defined in (Turi and Bit-Monnot 2022a): *Random (R)* that selects a random robot when needed, *First Available (FA)* that takes the first available robot, and *First Available with Longest Remaining Processing Time First (FALRPTF)* that prioritizes tasks regarding remaining processing time of packages. The first results have been obtained Figure 6 for small instances of the problem, in which only two packages should be processed. While they show that the system works with the new planning models and the integration of the planner, they do not prove that using a planner gives better execution paths than only resorting to reactive strategies. This can be explained first by the simulation itself that is not rich enough to represent system failures of robots or machines, or random arrival of new package to process. Adding these more realistic changes will worsen the *Random (R)* strategy and make the new ones more appealing. Therefore, an extension of the simulation to represent problems with more hazards would be a first step to better evaluate OMPAS. Secondly, the action duration time model is too simple, execution time for actions (e.g., moving around, or machining) takes a simple unit of time: one second. These actions should use the distance to travel, which would produce more accurate temporal plans. Last, currently the real execution in simulation is almost instantaneous, and thus the planning can hardly produce a plan before the state of the world changes. In the real world, most robot actions will take tens of seconds, and this will allow the planner to produce a plan to be used by the acting manager. These improvements in the simulation, the timing models and the execution time/planning time ratio should help us show the interest of the presented approach.

On a more positive note, in the given instances, the planner took a few milliseconds to find a first plan, and of a few seconds to obtain an optimal one. This confirms that the planner can be used online, although this should be confirmed on larger instances to assess its scalability.

7 Conclusion

OMPAS was first introduced by (Turi and Bit-Monnot 2022b), and later extended with concurrency and task interleaving (Turi and Bit-Monnot 2022a). The version we present here now includes a continuous planning component to improve deliberation. Along with the planner, an *Acting Manager* is proposed to ease the use of online planning in a reactive acting engine, using a unified structure to represent the execution traces, and the produced plans. Planning models are still generated by automatically translating the executed operational models as chronicles, allowing rich temporal representation of actions, in particular to represent concurrent subtasks inside methods. The procedure to analyze the operational models has been improved, by first resorting to a richer intermediate formalism to represent programs defined with the dedicated acting language SOMPAS, and second to support the translation of concurrency and resource acquisition primitives. Preliminary results on the Gobot Sim domain are presented. With the support of temporally richer planning models, the acting system could be guided by the planner, but produces mitigated results, which can be improved with a more realistic simulation, a richer temporal modelling of the domain, and some engineering work to finish the integration of the continuous planning module. Current work is addressing these issues, and seeks to integrate OMPAS on new robotic problems and then, comparing it to other approaches, assess its usability, scalability and genericity as an acting system.

8 Acknowledgments

This work has been supported by the European Union's Horizon 2020 research and innovation programme under grant agreement No. 101016442 (AIPlan4EU) and by the Artificial and Natural Intelligence Toulouse Institute - Institut 3IA (ANITI) under grant agreement No ANR-19-PI3A-0004.

References

Bansod, Y.; Nau, D.; Patra, S.; and Roberts, M. 2021. Integrating Planning and Acting With a Re-Entrant HTN Planner. In *HPlan Workshop, ICAPS*.

Beetz, M.; Mösenlechner, L.; and Tenorth, M. 2010. CRAM: A Cognitive Robot Abstract Machine for Everyday Manipulation in Human Environments. In *IEEE/IROS*. IEEE.

Bit-Monnot, A.; Ghallab, M.; Ingrand, F.; and Smith, D. E. 2020. FAPE: A Constraint-based Planner for Generative and Hierarchical Temporal Planning. *arXiv:2010.13121 [cs]*.

Cashmore, M. 2015. ROSPlan: Planning in the Robot Operating System. In *ICAPS*.

Chien, S.; Knight, R.; Stechert, A.; Sherwood, R.; and Rabideau, G. 2000. Using Iterative Repair to Improve the Responsiveness of Planning and Scheduling. In *AIPS*.

Clement, B. J.; and Durfee, E. H. 1999. Theory for Coordinating Concurrent Hierarchical Planning Agents Using Summary Information. In *AAAI/IAAI*.

Despouys, O.; and Ingrand, F. 1999. Propice-Plan: Toward a Unified Framework for Planning and Execution. In *European Workshop on Planning*.

Dvořák, F.; Bit-Monnot, A.; Ingrand, F.; and Ghallab, M. 2014. A Flexible ANML Actor and Planner in Robotics. In *PlanRob Workshop, ICAPS*.

Fox, M.; and Long, D. 2003. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *JAIR*.

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 *HPlan Workshop, ICAPS*.

Ingrand, F.; Chatila, R.; Alami, R.; and Robert, F. 1996. PRS: A High Level Supervision and Control Language for Autonomous Mobile Robots. In *IEEE/ICRA*.

Ingrand, F.; Lacroix, S.; Lemai-Chenevier, S.; and Py, F. 2007. Decisional Autonomy of Planetary Rovers. *Field Robotics*.

Li, R.; Patra, S.; and Nau, D. S. 2021. Decentralized Refinement Planning and Acting. In *ICAPS*.

Martin, F.; Clavero, J. G.; Matellan, V.; and Rodriguez, F. J. 2021. PlanSys2: A Planning System Framework for ROS2. In *IEEE/IROS*. IEEE.

McGann, C.; Py, F.; Rajan, K.; Thomas, H.; Henthorn, R.; and McEwen, R. 2007. T-REX: A Model-Based Architecture for AUV Control. In *Workshop on Planning and Plan Execution for Real-World Systems*.

Muscettola, N.; Dorais, G. A.; Fry, C.; Levinson, R.; Plaunt, C.; and Clancy, D. 2002. IDEA: Planning at the Core of Autonomous Reactive Agents. In *ICAPS*.

Muscettola, N.; Nayak, P.; Pell, B.; and Williams, B. C. 1998. Remote Agent: To Boldly Go Where No AI System Has Gone Before. *Artificial Intelligence*.

Patra, S.; Mason, J.; Ghallab, M.; Nau, D.; and Traverso, P. 2021. Deliberative Acting, Online Planning and Learning with Hierarchical Operational Models. *Artificial Intelligence*.

Rabideau, G.; Knight, R.; Chien, S.; Fukunaga, A.; and Govindjee, A. 2000. Iterative Repair Planning For Spacecraft Operationsusing The ASPEN System. In *ISAIRAS*.

Sardina, S.; de Silva, L.; and Padgham, L. 2006. Hierarchical Planning in BDI Agent Programming Languages: A Formal Approach. In *AAMAS*.

Shivers, O. 1988. Control Flow Analysis in Scheme. In *ACM SIGPLAN*.

Steele, G. 1990. *Common LISP: The Language*. Elsevier.

Turi, J.; and Bit-Monnot, A. 2022a. Extending a Refinement Acting Engine for Fleet Management. In *ICTAI*.

Turi, J.; and Bit-Monnot, A. 2022b. Guidance of a Refinement-based Acting Engine with a Hierarchical Temporal Planner. *IntEx Workshop, ICAPS*.

Yao, Y.; and Logan, B. 2016. Action-Level Intention Selection for BDI Agents. In *AAMAS*.