Extending a Refinement Acting Engine for Fleet Management

Jérémy Turi, Arthur Bit-Monnot

Extending a Refinement Acting Engine for Fleet Management: Concurrency and Resources

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

***Abstract*—Recent years have seen an important increase in the complexity of deployed robotic systems, both in terms of the number of robots involved, and scale of the tackled problems. The key challenge in this context is to allow the design of fleet control systems that, on the one hand, allow flexible and reactive operation of individual robots and, on the other hand, enable the system to optimize the global behavior of the fleet in order to increase its effectiveness and efficiency. To approach this problem, we propose to extend the Refinement Acting Engine (RAE) that has been used to program the behavior of autonomous agents through a hierarchical decomposition of high-level tasks into primitive commands, and is the subject of active research in order to guide its decisions with planning and scheduling techniques. The core of our proposal is to provide first-hand support for concurrency in the RAE procedure, allowing a natural representation for concurrent systems by reasoning on resource allocation. The resulting acting engine exploits a custom language that is designed to ease its integration with planning engines, both through its simple and orthogonal core constructs as well as in the explicit identification of decision points in the system operation. We provide an initial validation of the system in simulation on a logistic problem involving a fleet of robots.**

***Index Terms*—Acting, Multi-agent systems, concurrency, planning**

I. Introduction

Recent years have seen an increased interest in the deployment of fleets of robotic systems. Among many potentially impacted fields, Industry 4.0 is an area where the deployment of autonomous mobile robots (AMR) has the potential to increase both the flexibility and the efficiency of production lines [1]. However, the operation of such fleets comes with many challenges. Consider for instance a scenario where a fleet of AMRs is used to fulfill a set of tasks involving the manipulation and transportation of packages that need to be submitted to a series of processes at different workstations in a factory floor. In this context, the fleet management system must allow the continuous operation of the fleet to treat incoming requests, allocate tasks to the AMRs, handle possible contingencies (robot failure, unavailability of a workstation, etc.). Beside these operational requirements, the efficient exploitation of the fleet is tightly coupled with the ability of the system to efficiently allocate the resources (robots, workstations) and schedule the execution of the tasks.

Such systems thus require a combination of *(i)* reactive skills, in order to adapt the activity of individual robots or of the entire fleet to new requests and contingencies, and of *(ii)* explicit deliberation targeting the optimization of the global behavior of the fleet, in order to, e.g., minimize the exploitation costs or fulfill deadlines. In robotics, such scenarios have been traditionally handled by having an acting engine in charge of controlling a robot or a fleet with ad-hoc mechanisms, and procedures to react to changes in the environment or in the objectives. These acting systems can often be coupled with some sort of planning engine that can either provide a high-level strategy, or guide the operation of the system.

In this paper, we propose an extension of the Refinement Acting Engine (RAE) [2], an acting system that relies on the hierarchical refinement of high-level tasks into lower-level tasks and commands, based on a set of executable procedures. The objectives of this extension are to *(i)* provide native support for concurrency in the system, *(ii)* explicitly model resources, and *(iii)* clearly identify decision points in the operation model. Identification of decisions should facilitate guidance of the acting engine's choices by, e.g., automated planning techniques. Beside the adaptation of the original RAE execution procedure, we propose a dedicated language to specify executable procedures, with support for asynchronous execution, resource management and decision points. The language is a Lisp dialect with a limited set of primitives, which eases the automated analysis of operational model in order to use dedicated tools to guide the acting engine for decision points.

II. Related work

Several acting systems exist in the literature that propose different approaches to deliberative architecture and uses different languages to define the agent behavior.

Language-based systems such as RAE [2] and the Procedural Reasoning System (PRS) [3] are based on the iterative refinement of tasks into executable procedures, composed themselves of commands or tasks. The refinement is done at runtime, which avoids having to generate the full command sequence before execution. While PRS is a goal-oriented system, RAE is task-oriented, which is closer to Hierarchical Task Networks (HTN), and should make the integration of HTN planners more natural. Propice-Plan [4] extends PRS, and endows it with anticipation capabilities and continuous planning, from which our work is inspired. RAE has been extended with look-ahead capabilities such as UPOM [5], that uses an anytime planner to find the best refinement for a task in the current state thanks to rollouts in a Monte Carlo Tree Search (MCTS) enhanced by learning techniques.

However, UPOM does not ensure a valid plan as opposed to the Run-Lazy-Refineahead algorithm [6]. A recent work presented OMPAS [7], a system based on RAE in which planning models are extracted from operational models defined with a Lisp dialect, that we extend in our work. Another work proposed Dec-RPAE [8], a decentralized version of RAE that differs from our centralized view of fleet management.

Plan-Exec is a different approach to deliberation that proposes to first generate a plan and then execute and monitor it. T-REX [9] is a system based on IDEA [10], a descendant of RAX-PS [11] which differs in proposing a unified architecture for acting and planning based on tokens, that T-REX makes more robust by explicitly defining tokens ownership. However, the programming of an agent is done with a constraint-oriented language, which can be difficult to use compared to general purpose languages. Some systems such as IxTet-Exec [12] and more recently FAPE [13] use a Plan-Exec architecture using using temporal planners to take into account time.

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 in [14] to generate deadlock-free interleaving of plans. An interleaving of plans at the command level is proposed in [15] using MCTS, which improves performances compared to Round Robin, but does not ensure deadlock-free interleaving.

III. Overview of the system

A. RAE, a hierarchical approach to deliberative acting

Several acting systems are based on the RAE algorithms as first defined in [2]. The objective of these systems is to carry out one or multiple *tasks* until they succeed, reactively handling contingencies and execution errors occurring during the execution. To meet this objective, RAE-based systems refine down high-level tasks into a set of elementary *commands*, that are executable by a platform (e.g. a robotic system) at runtime.

The refinement is based on the hierarchical representation of the capabilities of an agent defined as a tuple (A, T, M), where A is the set of commands, 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 through lower-level tasks and commands. A method $m \in M$ is associated to 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. An example of method is given in Figure 1. For each task it faces, RAE refines it by finding an applicable method, which means that pre-conditions are true in the current state ζ, and executes the corresponding body. In case the method fails, RAE searches another method to try until a method succeeds, or no more methods are applicable. In case a command needs to be executed, a request is sent to the platform and RAE awaits the result of the command, which can either be *success* or *failure*.

```
m-transport(o, l, t)
  task: transport(o, l)
  pre-conditions: fuel(t) ≥ 50%, capacity(t) ≥ 20%
  body:   (begin
              (define loc−o (read−state loc o))
              (define loc−t (read−state loc t))
              (if (!= loc−o loc−t)
                  (exec drive t loc−o))
              (exec load t o)
              (exec drive t l)
              (exec unload t o))
```

Fig. 1: An example of RAE method for a logistic task to transport an object o to a location l. The body of the method is a Lisp expression which evaluation depends on the initial location of the truck.

B. Revision of RAE

The first definition of *RAE* proposed in [2] is limited in several aspects: the progression of concurrent tasks relies on *Round Robin*, advancing one instruction at a time for each task, and methods are limited to a sequence of instructions.

The *Operational Model Planning and Acting System (OMPAS)* is a revision of *RAE* that extends the deliberation capabilities of the acting engine. It uses a new acting language for operational models that was specifically designed for the automated analysis of operational model, in order to use planning techniques to guide RAE. A specific definition of concurrency inside methods is provided, with a formalism for resources shared among several concurrent tasks, on which the acting engine can reason to improve the progression of multiple tasks using the same resources. Moreover, the programmer can identify specific decision points where the acting engine has some decision freedom that can be exploited to optimize the overall behavior of the system.

To implement those changes the architecture and operation of the system have been adapted, and an overview is given in Figure 2. The essential difference with the original RAE lies in the fact that each top level task submitted by a user is started in a dedicated thread and has access to synchronization primitives, relieving the acting engine from explicitly orchestrating the execution.

C. Operation of the System

OMPAS can be seen as a complete system that embeds an acting system interfaced with a robotic *platform*, used to execute and monitor several tasks in parallel. A platform must provide an interface to execute or cancel commands, as well as information about the perceived world and command execution status. When a command should be executed, a request is sent to the platform and the acting engine monitors its status. The command is first in a *pending* status, and switches to the *running* status if the command is accepted by

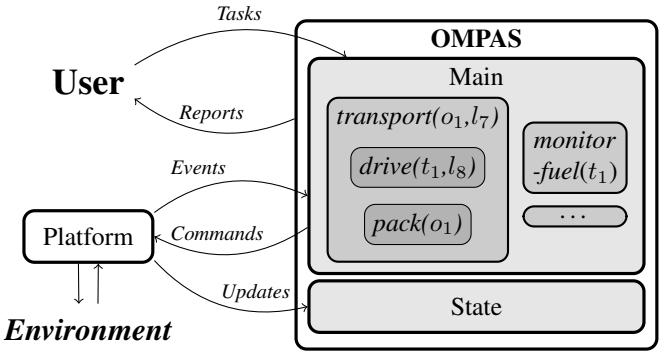

Fig. 2: A high-level view of the operation and architecture of OMPAS. The *user*, either an operator or a program, can send new tasks to be executed. A new task is processed by the *main* thread, which creates a new thread to handle the execution of the task. The user receives task status reports and platform status updates.

the platform, otherwise the command is a *failure*. This initial failure can happen if, e.g. incompatible commands have been requested at the same time. Once the platform has completed the execution, it communicates back a *success* or *failure* status depending on its outcome, and the command request is considered completed.

The execution of a task resorts to the EXEC-TASK function presented in Figure 3 that first generates all *applicable* methods in the current state ζ, and then SELECT-METHOD arbitrarily selects an untried method among them. Several techniques can be used to select a method, the default selection will choose the first method of M_{app}. The returned method is a program that is executed with the EXEC-BODY function. If the result of the execution is a failure then the acting engine tries to execute the task again with a limited set of possible methods, restricted by the methods that have been already tried. The function EXEC-TASK is called until either a method is a *success*, or no method is applicable.

While the overall system remains simple, most of the complexity of the acting engine lies in the EXEC-BODY procedure that is in charge of interpreting a user-defined program specifying the behavior of the method. The method's body uses a Scheme dialect that allows the usage of general-purpose programming constructs (e.g. branching, loops, arithmetic) together with acting-specific features, notably to enable the execution of commands or tasks as well as the acquisition of resources.

IV. ACTING LANGUAGE

In this section, we propose Scheme OMPAS (SOMPAS), a *Lisp* dialect, and in particular a variation of *Scheme*, that provides (i) generic constructs to define the behavior of an agent, such as loop, branching, and error recovery, (ii) acting primitives to query the state of the system and request the execution of a task or command, (iii) the ability to handle concurrent execution and shared resources management.

procedure EXEC-TASK(τ, M_{tried})
  $M_{app} \leftarrow$ APPLICABLE-METHODS(τ, ζ)
  $m \leftarrow$ SELECT-METHOD(τ, $M_{app} \setminus M_{tried}$)
  if $m = \varnothing$ **then** ▷ No untried applicable method left
    return failure
  $res \leftarrow$ EXEC-BODY(m)
  if $res = failure$ **then** ▷ Retry, with m forbidden
    return EXEC-TASK(τ, $M_{tried} \cap \{m\}$)
  else
    return res

Fig. 3: Adaptation of RAE's procedure for executing a task τ. The procedure arbitrarily selects a method m that is applicable in the current state ζ and has not been previously tried.

The first key feature of the language is to explicitly identify where the interpreter (i.e. the acting engine) has some freedom of decision, which gives some slack to the system in order to optimize its global behavior. The second key feature is to facilitate the automated analysis of programs in order to predict the behavior of the system following a decision by restricting the core language to fewer primitives than *Common Lisp*, the primitives *begin, if, define, lambda, quote* being sufficient to define a program. More complex programming constructs can be added by defining macros from the composition of those primitives. Like the original Scheme language that heavily builds on the lambda calculus, the core of our language is purely functional, which notably forbids mutation and side effects.

A. Concurrency and interruption

In order to be able to define concurrency, we adapted the *Scheme* dialect by adding new types and primitives to the language. As a reminder, the execution of *Scheme* programs is based on the recursive evaluation of LValues, that can be either an *Atom (boolean, number, procedure, symbol)* or a *List*. With the following additions, it is possible to evaluate an LValue in a new thread, await its result, or interrupt it. The first thing we define is the *handle*, a new kind of LValue that represents the thread executing the asynchronous evaluation. A *handle* can be manipulated with the following functions:

- *(async e)* starts a new thread where the expression e will be evaluated and immediately returns the *handle* of this thread.
- *(await h)* takes as argument a *handle*, and awaits the end of the corresponding thread. When this occurs, the *await* expression returns the result of the expression that was evaluated in the thread.
- *(interrupt h)* takes as argument a *handle*, and sends an interruption signal to the concurrent evaluation. Then, the function awaits the result of the interrupted evaluation. When an interruption signal is sent to a concurrent evaluation of a handle, the interruption signal propagates recursively to all expressions currently evaluated in the handle.

B. Acting primitives

The first purpose of an acting language is to control the behavior of a robotic platform and leverage the features of the acting engine. The following functions provide interfaces with both the platform and the acting engine:

- *(exec a $p_1 ... p_n$)* executes and monitors the execution of a task or command a with parameters $(p_1, ..., p_n)$. If a is a *command*, the interpreter resorts to the platform to execute the command and awaits on its result. The interruption of a command provokes a cancel request to the platform. In the case where a is a *task*, RAE calls the EXEC-TASK algorithm of *OMPAS* and awaits its result. An interruption signal on the execution of a task is propagated to the body of the method, and therefore should be handled by the method.
- *(read-state sf $p_1 ... p_n$)* is used to get the actual value of a state-variable designated by a state-function *sf* and a list of parameters $(p_1, \ldots, p_n)$.
- *(arbitrary set λ)* selects an arbitrary element e_i from a set $\{e_1, ..., e_n\}$. The acting engine is free to select any variable in the set. The optional function λ that returns an element of the set can be used to suggest a value to use (suggestion that the engine is allowed to ignore).

More advanced features and functions are presented in the documentation of the system at https://plaans.github.io/ompas/.

C. Resources

As soon as concurrency is involved, management of shared resources becomes a critical aspect. The resource design that we propose here attempts at fulfilling two needs: provide exclusive resource usage, and let the acting engine define its allocation strategy.

We define a resource as an object with an initial capacity C_{init}. A resource r can be acquired at time t with an amount c that is lower than or equal to the current capacity C_t. Upon acquisition, the acting engine ensures that no race-condition occurs that would result in an over allocation and the capacity is immediately decreased by the amount c. We distinguish *unary* and *divisible* resources. A *unary* resource can be acquired by only one task at a time, where initial capacity and requested amount are always one. A *divisible* resource with an initial capacity C_{init} can be acquired with any $c_t \in [0, C_{init}]$. At the difference of real-time systems and mutexes, there is no guarantee on the order of access to resources, as it defers this decision to the acting engine and reasoning systems. When a resource is released, its capacity is increased by corresponding amount.

The declaration of a resource is done with the function *(new-resource r C)* that takes the label *r* of the resource, and an initial capacity C for a *divisible* resource. The acquisition of a resource r is done with the function *(acquire r c)* that takes as parameter the label of the resource that is requested, and the quantity c needed if r is *divisible*. Once the acquisition has been validated by the system, the function returns a *resource-handle* h. If h goes out of scope, the resource is automatically released. A resource can be explicitly released thanks to the *(release h)* function, that takes as parameter h. The acquisition of a resource can be interrupted to avoid blocking a program waiting too long on a resource.

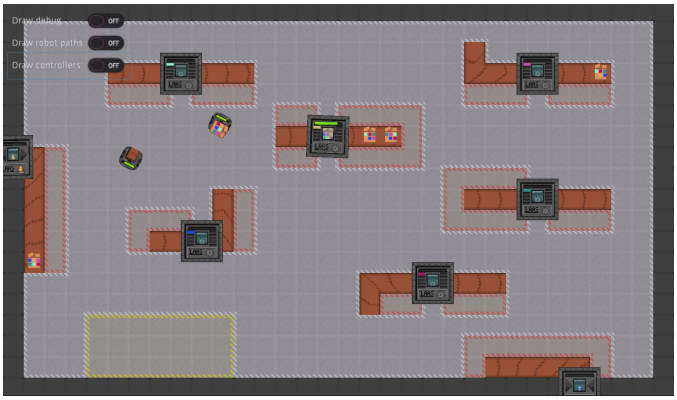

Fig. 4: Overview of a 6×6 job shop scenario in *Gobot-Sim* composed of one *input* machine (on the left) that feeds the environment with unprocessed packages, six *processing* machines that can do a predefined process, and one *output* machine (at the bottom right) that receives fully processed packages. Two robots can be used to dispatch packages on the machines. The recharge area (in yellow) is available at the bottom.

V. EXPERIMENTAL RESULTS

To show the relevance of the presented approach, we present here both the successful integration of *OMPAS* with *Gobot-sim*, a new job-shop benchmark for acting systems. *Gobot-sim* is simpler than RoboCup Logistics League Simulation [16]. *Gobot-sim* is similar to Craftbots [17], but focuses on logistic problems similar to *Job Shop Scheduling Problems (JSSP)* [18]. It is composed of a fleet of holonomous robots that can be controlled to transport packages between machines used to process packages. An example is visible in the Figure 4.

The robots can manipulate one package at a time thanks to the *pick* and *place* commands, and can be moved with *do-move* for precise displacements, *go-charge* and *navigate-to-area*. Other commands are available, but are not relevant for the present work. The robots' energy comes from batteries that discharge continuously unless robots are at recharge locations. The state is composed of information concerning the robots, the machines and the topology of the environment, e.g. locations of machines. The state is considered to be fully observable in the current version.

Each package should be processed on all machines in a given order, and each machine can only process one package at a time, which means two jobs cannot be done on the same machine at the same time, but a package can wait at the entrance of the machine. The role of the acting engine is to control the fleet of robots in order to minimize the time to process all packages. We here assume that all packages and

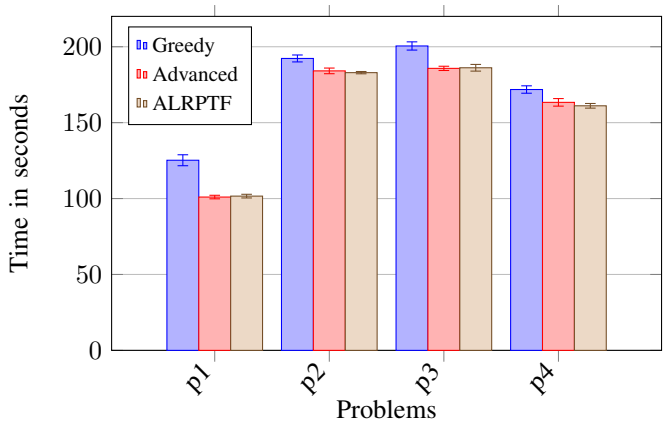

Fig. 5: Comparison of the mean time to execute 4 6x6 job-shop problems with three different allocation strategies. Each pair problem-strategy has been executed 10 times. Timescale of the simulator is set to 4.

robots are created at the beginning of the benchmark. The acting model is composed of a unique high-level task, that does the following:

- For each package p in the system, a new task *process-package* p which goal is to process the package is executed in a new thread.
- For each robot r in the system, a new task *monitor-battery* r to monitor the battery level of r is executed in a new thread. Each time the battery is under a critical level (here 40%), r is sent to recharge its battery.

In such representation of the problem, we can distinguish two kinds of unary resources: robots and machines.

The role of the acting engine is therefore to allocate robots to packages, and schedule the passage of packages on the machine. In the current version of the system, the acquisition of a resource is sorted by a priority level, where the task that monitors the battery of a robot has the highest priority. Several strategies that differ both on the choice of the resource and the priority are compared in Figure 5. The *Greedy* strategy acquires the machine and an arbitrary robot with no priority between packages. The *Advanced* strategy improves the choice of the robot by acquiring the first available robot, thanks to a race to acquire each robot, the first acquisition cancelling the others. The *Advanced with Longest Remaining Processing Time First (ALRPTF)* strategy gives priority on the machine and the robot to packages with the most remaining processing time. We can see that the *Advanced* strategy is always better than the greedy one. Results of the *ALRPTF* strategy show a small additional performance gain. For this kind of problem, it would be particularly interesting to use a planning or scheduling system to guide resource allocation and compare with those reactive strategies.

VI. Conclusion

The present work proposes an extension of the RAE procedure to handle tasks in a concurrent environment. The existing RAE procedure allows us to define reactive programs to handle complex problems in a hierarchical fashion. We add first-class concurrency support, which greatly simplifies the core of the acting procedure by removing the need for ad hoc management of concurrently executing tasks. Operational models are defined with a dedicated acting language, that integrates functions to define concurrent programs. Shared resources are explicitly represented, which allows the system to deliberately interleave the progression of several tasks depending on the available resources. The developed system has been integrated in the *Gobot-Sim* benchmark, that simulates the operation of a robotic fleet in a logistic environment where the support for concurrency and resource management turned out to be essential for a correct operation of the fleet.

References

[1] G. I. Fragapane, D. A. Ivanov, M. Peron, F. Sgarbossa, and J. O. Strandhagen, "Increasing flexibility and productivity in Industry 4.0 production networks with autonomous mobile robots and smart intralogistics," *Annals of Operations Research*, 2022.

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

[3] F. Ingrand, R. Chatila, R. Alami, and F. Robert, "PRS: A high level supervision and control language for autonomous mobile robots," in *Proceedings of IEEE International Conference on Robotics and Automation*, 1996.

[4] O. Despouys and F. F. Ingrand, "Propice-Plan: Toward a Unified Framework for Planning and Execution," in *Recent Advances in AI Planning*. Springer Berlin Heidelberg, 2000.

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

[6] Y. Bansod, D. Nau, S. Patra, and M. Roberts, "Integrating Planning and Acting With a Re-Entrant HTN Planner," *ICAPS Workshop on Hierarchical Planning (HPlan)*, 2021.

[7] J. Turi and A. Bit-Monnot, "Guidance of a Refinement-based Acting Engine with a Hierarchical Temporal Planner," *ICAPS Workshop on Integrated Planning, Acting, and Execution (IntEx)*, 2022.

[8] R. Li, S. Patra, and D. S. Nau, "Decentralized Refinement Planning and Acting," *Proceedings of the International Conference on Automated Planning and Scheduling*, 2021.

[9] C. McGann, F. Py, K. Rajan, H. Thomas, R. Henthorn, and R. McEwen, "T-REX: A Model-Based Architecture for AUV Control," *3rd Workshop on Planning and Plan Execution for Real-World Systems*, 2007.

[10] N. Muscettola, G. A. Dorais, C. Fry, R. Levinson, C. Plaunt, and D. Clancy, "Idea: Planning at the core of autonomous reactive agents," *Sixth International Conference on AI Planning and Scheduling*, 2002.

[11] N. Muscettola, P. Nayak, B. Pell, and B. C. Williams, "Remote Agent: To boldly go where no AI system has gone before," *Artificial Intelligence*, 1998.

[12] F. Ingrand, S. Lacroix, S. Lemai-Chenevier, and F. Py, "Decisional autonomy of planetary rovers," *J. Field Robotics*, 2007.

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

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

[15] Y. Yao and B. Logan, "Action-Level Intention Selection for BDI Agents," *Association for Computing Machinery (ACM)*, 2016.

[16] T. Niemueller, E. Karpas, T. Vaquero, and E. Timmons, "Planning Competition for Logistics Robots in Simulation," *ICAPS Workshop on Planning and Robotics (Plan-Rob)*, 2016.

[17] L. Nemiro, G. Canal, O. Lima, M. Cashmore, and M. Roberts, "Designing an Adaptable Benchmark and Competition Simulation for Integrated Planning and Execution," *Workshop on the International Planning Competition (WIPC)*, 2021.

[18] D. Applegate and W. Cook, "A computational study of the job-shop scheduling problem," *ORSA Journal on computing*, 1991.