CLAPLEX
A Control Language and Architecture for Planning and Execution

Annita Vapsi, Daniel Borrajo,[*] Manuela Veloso
J.P. Morgan AI Research
{annita.vapsi, daniel.borrajo, manuela.veloso}@jpmchase.com

Abstract

Previous work on planning and execution has proposed a three-tiered architecture where a middle layer exists between planning-based generated plans, which define high-level abstract actions, and low-level commands connected to a real-world executor. This middle layer is usually termed as acting. Current proposals to implement the acting layer vary, from solutions utilizing declarative or programmatic language representations, to solutions that do not incorporate a planner, and ones that utilize hierarchical planning constructs and plan refinement frameworks. In this paper, we propose a control architecture, CLAPLEX, that utilizes a new programming language providing a simple interface to define the acting layer with classical execution-monitoring constructs. This language is seamlessly integrated with Python, a general purpose programming language, that adds total flexibility to creating acting code. We also propose an architecture that includes a compiler from plans to acting code in that programming language. We report on the successful use of this programming language to create an open control program for an office robot.

Introduction

Planning uses an agent's initial state and generates a sequence of actions with the intent of achieving the agent's goals. Thus, a key component of a comprehensive planning framework is the ability to execute plans. Most works in the literature discuss the connection between planning and execution (Ingrand et al. 1996; Simmons and Apfelbaum 1998; Guzmán et al. 2012; Pinover et al. 2020; Cashmore et al. 2015), with some (Patra et al. 2019b; Ghallab, Nau, and Traverso 2016) paying attention to the use of a three-tiered architecture that would be composed of a deliberative planning component, an acting component, and an executive. In particular, this paper focuses on the acting component. Our approach, takes as input a plan, executes each action in the plan and monitors the execution of the action (inter and post execution).

While architectures such as the one proposed in this paper have been mainly used to control robotic systems (Pinover et al. 2020; Schaffer et al. 2018), they are general enough to be used in other related domains, such as games, logistics, or workflow execution for business processes. Each of these types of applications require a different kind of executive. For instance, ROS (Macenski et al. 2022) is used for robotic tasks, while DMN (OMG 2021) is commonly used for workflow applications. Our work aims to provide an executive that is independent of the application it is being used in. Previous works have proposed many different ways to create the acting component (Verma et al. 2005; Ingrand et al. 1996; Ingham, Ragno, and Williams 2001). These approaches are very flexible in the kinds of acting code they can generate, but lack seamless integration with external libraries.

In this paper, we define a planning-execution architecture, CLAPLEX, based on the integration of deliberative planning with a programming language, ALA. CLAPLEX allows developers to easily deploy acting and execution code. The connection between planning and execution is realized by compiling plans into control programs in ALA, the CLAPLEX acting language, that includes the execution and monitoring of actions. Each action in the plan is translated to a primitive function call in ALA. The connection between CLAPLEX and any external system, such as ROS, responsible for executing the external system's primitive actions, is done by defining Python functions that can be embedded in the code of the ALA control program. Given the use of Python, the execution of the controller defined in CLAPLEX can be easily linked to any external software libraries and tools used to build applications in this external system. Our current approach allows us to be agnostic to the underlying executive-language. Changing Python as the executive language would not require any change in the language, and very little changes in the architecture.

In its current state, CLAPLEX, takes plans generated from classical planners and translates them into ALA. While CLAPLEX is currently designed for sequential deterministic plans, ALA already provides the ability to support partial-order, conditional or temporal plans.

We start by introducing CLAPLEX, the system architecture. We then introduce the ALA programming language, discussing design choices made for language semantics and providing the language syntax in EBNF format. We provide an example of an automatically generated program and describe how it has been integrated with an execution and mon-

[*]On leave from Universidad Carlos III de Madrid

itoring framework. We present the interpreter and describe how the language was used on a robot domain use case. Finally, we provide an account of related work.

Architecture

Figure 1 depicts the planning-execution architecture of CLAPLEX. The boxes in dashed outline represent input/output information, while the solid filled boxes represent modules. As seen in the figure, the `Planner` module takes as input the planning domain and problem files and solves the planning problem. The plan is then fed into the `Compiler` module to produce the ALA program.

The `Syntactic Analysis` module takes the program as an input, together with the definition of the ALA syntax in EBNF format, and produces the AST which is used together with the `Static Functions` module by the `Executive` to execute each action in the environment. The `Static Functions` module is a Python program that contains the definitions of primitive functions for the execution, as well as the functions responsible for regressing action preconditions to compute monitoring checks, and execute the monitoring actions. This module is imported at the top of the ALA program and is called whenever a primitive, or a monitoring action is requested. Finally, the `Executive` receives as input the domain and problem files which are used to compute the plan's regressed precondition, and state information from the environment to support the execution of the ALA program.

Next, we will present in more detail the components of the architecture. The CLAPLEX planning execution and monitoring framework is supported by five modules, the Planner, the Compiler, the Syntactic Analysis module, the Static Functions module and the Executive. Overall, the architecture takes as an input the planning domain and problem files and solves the planning problem. The plan is then automatically compiled into the corresponding ALA program in a domain-independent fashion. In generating the ALA code, the same module is also responsible for embedding monitoring checkpoints in the code. We provide Algorithm 1, which details the way in which monitoring checkpoints are included in the ALA code by appending strings to list L and eventually adding the strings to ALA code. These strings include the pieces of code responsible for regressing preconditions and executing monitoring checks and plan actions. The algorithm starts by adding to list L the regressed preconditions function and assigning it to variable p. Since the `Executive` takes as input the domain and problem files, they do not need to be added as parameters to the $get\_regressed\_preconditions$ function. Next, the algorithm loops over the number of action in the plan and appends to the list the command responsible for executing the current action. Finally, the algorithm adds a conditional statement command to the list of ALA commands that checks whether the monitoring function returns 'fail' for the current action and halts execution if that is true, else continues with the loop. We make the assumption that, by using the planning domain and current state, the $get\_regressed\_preconditions$ function is able to reconcile high-level planning actions/preconditions to their low-level state counterparts, to compute potential future failures.

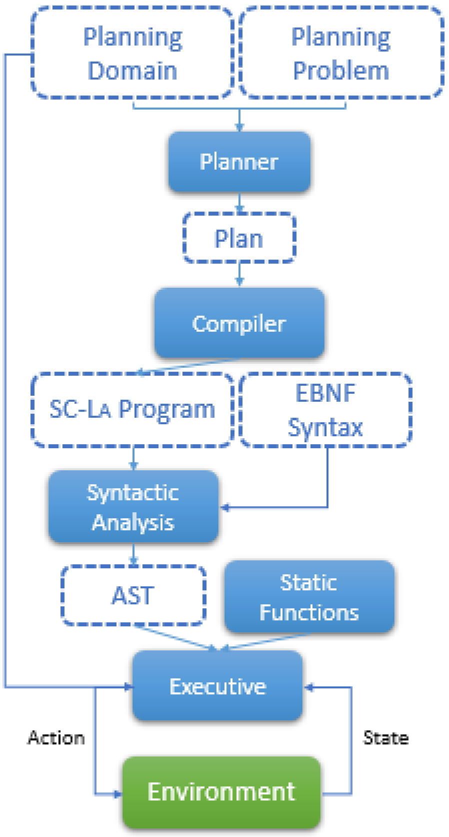

Figure 1: CLAPLEX planning-execution-monitoring architecture.

The monitoring checks are responsible for pulling the regressed preconditions of the actions to be executed, and assessing if the plan will fail at any of the future action steps. The functions used to monitor execution are placed in a Python script and imported at the top of the ALA program. Thus, the interpreter executes the actions in the plan while at the same time functioning as an execution monitoring module. As of now, our module halts the execution of the plan as soon as the regressed preconditions fail for the action at hand.

ALA- The Language

ALA is a high-level, modular, highly orthogonal programming language for control. There are currently two ways to use the language: 1) as an acting mechanism of a planning system, where the compiler's functionality is, for now, limited to translating sequential plans; and 2) as a programming language to directly model acting programs. ALA is more expressive than what the compiler supports. For instance, the language currently includes powerful functionality, like action parallelization which the compiler is not leveraging. As is, the compiler translates classical plans by using ALA's sequential execution mode. The following subsections pro-

Algorithm 1: Algorithm for embedding monitoring checks in the ALA code.

```
Input: Plan as a sequence of actions, $a = \{A_1 \cdots A_N\}$,
Output: List of execution-monitoring commands to be included in the control program
1: $L \leftarrow$ [‘$p = get\_regressed\_preconds()$’]
2: for $\forall i \in \{1, N\}$ do
3:     $L \leftarrow L+$ [‘$execute(A_i)$’]
4:     $L \leftarrow L+$ [‘if monitor($p(A_i)$) = fail then fail’]
5: end for
6: return $L$
```

vide details of the language’s functionality and semantics, its syntax and the development of the interpreter. At the end of the section, we provide an example of a ALA program utilizing a good proportion of the currently available ALA functionality.

Functionality and Semantics

A top-level execution mode specification, loops, if/while statements, function definitions and function calls are some of the constructs built into the ALA language capabilities. These, together with support for conditional branching, variable assignments, and mathematical and logical expressions make up ALA’s rich control architecture.

Additionally, Python has been integrated with the language in the form of function imports from separate Python scripts, promoting code simplicity and the ability to expand the language’s capabilities to all additional Python constructs and packages. Potential uses include, but are not limited to, the integration of the CLAPLEX control program with external systems, such as robots, and the use of primitive functions to call on the execution of actions in these systems.

The language has been developed to execute a set of actions, in one of three modes: sequential execution, parallel execution or any-order execution. These modes mimic the different plan control structures. While sequential execution allows for the simpler total-order type of plans, all actions called inside the ‘parallel’ execution mode keyword are executed concurrently using Python’s multiprocessing module. In this way, we are able to leverage a machine’s multiple processors to execute all actions at the same time. On the other hand, actions called inside the ‘any_order’ execution mode keyword are executed in a random sequence. Nesting of different execution modes is allowed.

Syntax

We provide the full EBNF definition of ALA in the Appendix. In this section, we provide an example of a plan produced by the planner and specify how it is translated to ALA code. The use case here involves a robot with moving and speech primitives. The plan solution is provided in Listing 1 and includes a move action from position `c1` to position `c2`, followed by a greeting action at `c2` that checks that the robot and human are in the same location and finally a second move action from `c2` to `c4`. In this example,

Listing 1: PDDL generated plan

```
(move c1 c2)
(greet-human p1 c2 c2 hi)
(move c2 c4)
```

Listing 2: Translated ALA code from PDDL code in Listing 1

```
import static_functions
program({}
sequence(
  p = regressed_preconds(),
  move('c1', 'c2'),
  if (monitoring(p(1)) == 'fail'):
    fail()
  else:
    greet_human('p1', 'c2', 'c2', 'hi'),
  if (monitoring(p(2)) == 'fail'):
    fail()
  else:
    move('c2', 'c4')))
```

parameters prefixed by the letter `c` signify locations, while the ones prefixed by `p` signify people. We utilize a compilation module to convert the PDDL solution to ALA code. The equivalent ALA code is included in Listing 2.

As shown in the automatically generated acting program, we import a module called `static_functions` at the top of the script. This is the Python module which specifies the primitives and is responsible for calling ROS to execute them. Then, the control part of the script starts. Curly brackets inside `program` designate allocated space for function definitions, which for simplicity could be transferred to a separate script and instead imported at the top. Right below, the control sequence is called and prefixed with a mode of execution keyword, in this case `sequence`. Inside the brackets of the execution mode keyword, the syntax allows us to use different control constructs like while-loops and if-statements. For this example, we simply call the appropriate primitive functions corresponding to the actions specified in the plan.

In generating this program automatically, the compiler takes the plan and uses Algorithm 1 to get a sequence of commands, which includes monitoring. The compiler also initially embeds a regressed preconditions command at the beginning of the program. Furthermore, monitoring checks, are added between each action in the plan. In this way, the plan’s regressed preconditions are computed at the beginning and the monitoring actions confirm that the preconditions of the actions in the remaining part of the plan are met. We use conditional statements dependent on the return values of the monitoring actions to decide whether to execute the next action in the plan. As shown in Listing 2, the program only allows the execution of `greet_human` if the first monitoring check does not return ‘fail’.

Interpreting ALA code

In order to execute code written in ALA, we built an interpreter to Python. We used ANTLR (ANother Tool for Language Recognition), (Parr 1992) a powerful parser generator. ANTLR supports the development of an executive in two ways. It can be used to check that the CLAPLEx program is syntactically correct, and builds the corresponding Abstract Syntax Tree (AST).

ANTLR takes as an input the EBNF syntax, provided in the Appendix, and generates parser, lexer and visitor files to support the interpreter development. Finally, ANTLR takes as an input a new CLAPLEx script, builds the AST and uses the interpreter to execute the commands in the code.

CLAPLEx- A Robot Domain Use-Case

We provide an example ALA program, which we call `robot_interact` to demonstrate the extent of ALA's functionality after some enhancements were made to allow for actions to be executed in parallel or in any order. We use the key words `parallel` and `any-order` to replace the keyword `sequence` in the original form of the language. We have not yet improved the `Compiler` module to support the automatic compilation of plans for parallel/any-order execution, but we plan to use standard ways of computing partial-order plans from total-order ones (Veloso, Pérez, and Carbonell 1990). The program is shown in Listing 3. While this program has been written manually, the execution is completely automated by interpreting this code and sending the corresponding commands to the robot through ROS.

The program was executed on a robot with motion, speech and object recognition primitives. Our aim was to utilize all of our robot primitives in this example and showcase the ability of the program to create open loops. Robot primitives are called using ROS (Macenski et al. 2022). ROS provides us with a set of libraries and tools useful for building robot applications. We additionally used Python modules, such as *time* and *random*, that support execution. We halt execution using the *time* module to let the robot interact more fluidly with the environment while the *random* module's random number generator function is used to choose a random angle for the robot to turn when it finds an obstacle. ALA empowers the seamless integration of Python and ROS in this example by utilizing external modules and connecting and executing primitive actions on the robot while at the same time demonstrating the language's simple syntax for system control.

At the start of the program, we use a sequential execution mode to assign a value to variables *v* and *h*, Boolean variables for obstacle detection and human recognition. Then, we use a parallel execution of sensing primitives for human recognition and obstacle detection, forward motion and speech. As soon as the robot detects an obstacle, it stops moving forward and chooses a random angle to turn to before continuing execution. At the same time, the robot greets a person as soon as it detects one.

Figure 2 shows nine video frames of the robot performing this task. The robot is originally placed facing the wardrobe. As seen in the figure, as soon as the robot starts moving it sees the wardrobe and turns to the right. The robot avoids the wardrobe a second time between frames 3 and 4 and starts moving towards the camera. As the robot moves towards the camera, it recognizes a human behind the camera and says *Hello*. It then turns and moves away from the camera because it recognizes the human as an obstacle.

Listing 3: ALA control program for `robot_interact`

```
import static_functions
program({}
sequence(
    v = 0,
    h = 0,
    parallel(
        while (true):
            (h = human_check(),
             v = obstacle_check()),
        while (true):
            if (v == 0):
                forward(0.3)
            else:
                (r_a = random_angle(),
                move(0, r_a),
                wait(2)),
        while (true):
            if (h == 1):
                (speak('Hello'),
                wait(15)))))
```

Related Work

Multiple works on the integration of planning and acting have been explored in the literature. One may find characteristics of the CLAPLEx control and execution language in prominent languages including PLEXIL (Verma et al. 2005), RMPL (Ingham, Ragno, and Williams 2001) and TDL (Simmons and Apfelbaum 1998). Modularity, expressivity and syntax simplicity are shared by all four languages, with additional common characteristics listed below. Other works provide solutions vastly different to the proposed framework, but still aiming at solving this discrepancy between high level abstraction and low level execution (De Benedictis et al. 2022).

PLEXIL's tree like structure which includes action leaf nodes and expandable control nodes in the tree, is equivalent to ANTLR's native tree like structure, available to the user and implicitly embedded to the ALA interpreter. The distinguishing aspect of ALA is that the code does not include embedded tree like programming constructs and is written in a simpler language format, easily interpreted by new users. The ALA non-declarative, programmatic representation has allowed the language to be fully integrated with Python libraries. An additional advantage to PLEXIL is that the code fed to the interpreter is readable, as opposed to XML code sent to PLEXIL's Universal Executive (as recognized by its authors (Verma et al. 2005)). Just as PLEXIL, ALA can be utilized as a standalone execution-only system.

RMPL supports similar constructs to ALA, including conditional branching, iteration, parallel composition, sequential ordering and preemption (Ingham, Ragno, and Williams 2001). However, while RMPL code specifies the state to be achieved and not the action to be executed, CLAPLEx actions are decided by a planning module. RMPL's con-

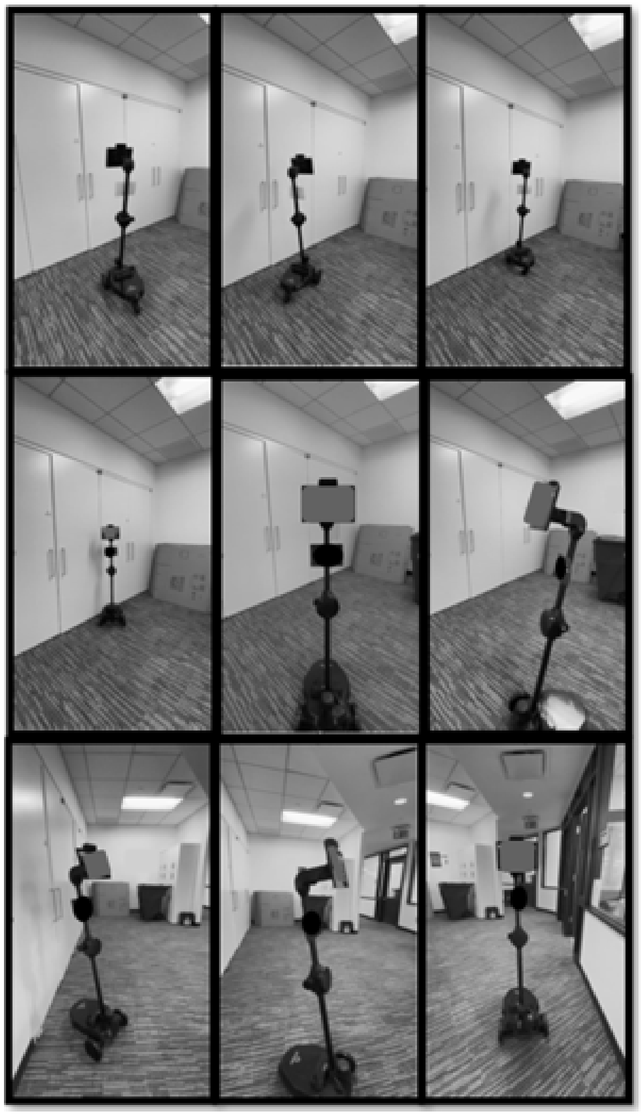

Figure 2: Video frames of robot executing the code in Listing 3.

straint based modeling and language constructs like parallel and sequential execution, and IF-THENNEXT-ELSENEXT, UNLESS-THENNEXT, WHEN-DONEXT, WHENEVER-DONEXT, DO-WATCHING control constructs, are simplified by ALA's if-else, for/while loop statements and keywords like parallel/sequence. Thus, ALA covers the extent of the RMPL functionality, but using a simpler language.

TDL and ALA differ in how the resulting control code is executed. While TDL is compiled to C++ code before execution (Simmons and Apfelbaum 1998), ALA is interpreted and executed in Python. The TDL utilization of a C++ platform-independent task-management library is comparable to ALA 's ability to utilize functions and primitive calls held externally in separate scripts and imported to the executing script, promoting clean scripting and code modularity. In the CLAPLEX case, this library exists in two modes, ALA code, or Python code. The functions contained in the scripts can include from simple primitive action execution to complex control and monitoring structures. Just as the CLAPLEX- Python integration, the TDL - C++ integration encodes the scripted code into a Task-trees. While TDL includes a elaborate structure on task tree execution flow, ALA's use of ANTLR, an off-the-shelf parser generator, means that there is no need to explicitly develop an AST or its traversal logic.

Behavior Trees are the de facto standard in the gaming industry for use in the control structure and behavior of Non-Player Characters and for robotic manipulation (Colledanchise and Ögren 2017). Our choice of embedding function return values in the ALA syntax and the ANTLR parser generator, creates a programming language with control structure and semantics resembling the execution flow of behavior trees (top to bottom and left to right execution), promoting modularity and supporting two-way control transfers (Colledanchise and Ögren 2017). In the case of TDL, its functions have no return values and thus, two-way control transfers are not supported.

PRS is another control and monitoring language with a declarative representation syntax supporting reactive reasoning capabilities and the ability to be asynchronously connected to low-level modules (Ingrand et al. 1996). We can draw a parallel between ALA and PRS in that while ALA is not represented declaratively, it does indeed support reactive reasoning, because of its rich control constructs and its seamless integration in Python. The two languages differ in that PRS does not incorporate a planner, rather calls a library of pre-specified plans and chooses among the ones most suitable for the goal at hand.

The STP architecture is a hierarchical control framework designed for adversarial multi-agent environments, with a specific focus on robot soccer (Browning et al. 2005). Although the architecture does not delve into the specifics of the acting layer, it provides a comprehensive description of the control framework's unfolding process, spanning from high-level goals of a robot team to the individual actions and action parameters of each robot within the team. While our current approach does not encompass an adversarial or multi-agent environment, we consider this work useful for future work, as it offers insights into unpackaging complex actions into simpler primitive controls, drawing inspiration from concepts such as long-term goal determination and strategic planning.

The ROGUE architecture combines high-level planning and low-level execution to create an autonomous robotic agent (Haigh and Veloso 1997). ROGUE acts as a task scheduler, handling asynchronous requests from multiple users. The architecture incorporates PRODIGY as the planning system, enabling real-world execution by mapping planned actions to robot primitives. Monitoring capabilities utilize sensing to detect differences between the expected and actual states, updating domain knowledge and refining future actions in the plan to achieve goals. Our current approach to monitoring halts execution as soon as an action fails and this would be one of the options to consider for potential future work on plan refinement.

Multiple papers in the literature call on plan refinement methods as being useful in control, execution and monitoring frameworks. For instance, Propice-Plan uses ideas like anticipation planning and plan synthesis to derive and adapt plans in cases of implicit goal specification or in cases of execution failure (Despouys and Ingrand 1999). ROSPlan (Cashmore et al. 2015), uses plan post-processing and validation, as well as system alerts in the case that the environment changes or the action fails, invalidating the current plan and forcing the system to replan. Instruction Graphs package an interactive approach to model verbal or written commands in a graph structure and allow developers to refine parts of the plan by correcting the parameters of actions or replacing an action altogether (Mericli et al. 2014). RAP allows for the simulation of different actions in the system to measure their impact, and chooses the appropriate action to modify the plan (Pinover et al. 2020). Similarly, both APE-Plan and RAE-Plan use Monte-Carlo simulation roll-outs of the relevant operational models for planning and plan refinement (Patra et al. 2019b,a). UPOM provides a solution to search this operational model space and make decisions about finding a near-optimal method for the current action (Patra et al. 2020). Hy-CIRCA focuses on providing a control structure for hybrid systems and combines the ideas seen in multiple other papers on plan refinement, hierarchical planning and an execution engine for coordination and replanning (Goldman et al. 2016). Similarly, (Turi and Bit-Monnot 2022) uses hierarchical operational models and a RAE implementation for plan refinement. This paper analyses the operational model and extracts planning domains from acting domains to select the best model for the task at hand. Further to plan refinement methods, Mendoza *et al.* (2015) introduces an execution monitoring framework which identifies and corrects discrepancies between the expected model of the world and the experienced reality, as a result of situations in which effects of actions in the real world deviate from the effects that the planner expected.

While we recognize the problem posed by APE (Patra et al. 2019b) on reconciling descriptive actions and operational models with rich control structures, our initial objective was to create a language that includes expressive constructs with rich control while fully integrating the language with a high-level programming language. To solve this reconciliation issue, APE uses the operational model for both acting and planning, which we defer to later research. Currently, CLAPLEX uses a planning module to generate plans and a compilation module to automatically generate ALA acting code.

Finally, ROSPlan, (Cashmore et al. 2015) reconciles the solution of a PDDL planner, generated on an abstract model of the world, with ROS, a system that calls on a robot's low level controls. Our robot domain use-case also uses ROS to execute robot primitives. The two approaches differ in that CLAPLEX has been developed to be able to call any library supported by Python, not only ROS. We could therefore use CLAPLEX with any other software supporting robotics as well as any other type of applications, as long as the execution system can be called within Python.

Conclusions and Future Work

In this paper we have introduced CLAPLEX, a System Control, Planning and Execution programming architecture backed by ALA, its control language, which aims to bridge the gap between planning, acting and execution. The language syntax and semantics resemble the compactness of a program written in Python, while keeping a top-level control structure, and potentially more complex and fluid inner architecture. We have additionally provided the language syntax in EBNF together with some information on how we built our interpreter using ANTLR.

In the future, we would be interested to explore potential extensions of the current monitoring framework to reason about other replanning strategies while executing. Additionally, recent work describes how Monte Carlo rollouts can be used for plan refinement (Patra et al. 2019b,a). This technique could allow us to drop the PDDL solution to ALA program translation and bridge the gap in abstraction between our declarative planning solution and programmatic execution. We would additionally be interested in developing the current architecture to support partial-order, conditional and temporal plans which would constitute a natural extension of the architecture's current methodology for including contingent monitoring checks for action failures. Finally, we would like to develop our current infrastructure to support the automatic compilation of parallel and any-order plans to resemble our current ability to automatically generate ALA code from sequential plans.

Acknowledgements

This paper was prepared for informational purposes by the Artificial Intelligence Research group of JPMorgan Chase & Co. and its affiliates ("JP Morgan"), and is not a product of the Research Department of JP Morgan. JP Morgan makes no representation and warranty whatsoever and disclaims all liability, for the completeness, accuracy or reliability of the information contained herein. This document is not intended as investment research or investment advice, or a recommendation, offer or solicitation for the purchase or sale of any security, financial instrument, financial product or service, or to be used in any way for evaluating the merits of participating in any transaction, and shall not constitute a solicitation under any jurisdiction or to any person, if such solicitation under such jurisdiction or to such person would be unlawful.

Appendix

The following listings include the complete EBNF syntax of ALA.

Listing 4: EBNF syntax definition for CLAPLEX- Table 1

```
imports : imp program;
imp : IMPORT ((COMMA)? file_name)*;
program : PROGRAM LPAR
            LCPAR
            r_functions
            RCPAR
            ((COMMA)? execute_mode)+
            RPAR;

r_functions : (r_function)*?;
execute_mode : EXEC_MODE  sentences;
sentences : LPAR ((COMMA)? sentence)+
    RPAR ;
sentence : var_assign|r_while|r_if|r_for
    |r_case|execute_mode|func_call ;

r_while : WHILE cond=condition_block
    COLON (sent_mult=sentences|sent=
    sentence) #whilestmnt;
r_if : IF condition_block COLON (
    sentences|sentence) (ELSEIF
    condition_block COLON (sentences|
    sentence))* (ELSE COLON (sentences|
    sentence))? ;
r_for : FOR iter_var= var IN lst=r_list
    COLON (sent_mult=sentences | sent=
    sentence) #forstmnt;
r_case : CHECK mathematical_expr COLON
    LPAR (CASE mathematical_expr COLON (
    sentences|sentence))+ RPAR ;

condition : left=condition op=andor
    right=condition #condnested
        | LPAR NOT cond=condition RPAR #
          condnot
        | comparison_expr #condcomp
        | bool_lit #condbool;
action : logic_expr #actlog
        | comparison_expr #actcomp
        | var_assign #actvar
        | mathematical_expr ~(SUB) #
          actmath;

action_block : ((COMMA)? action)+ #
    blockact ;
condition_block : LPAR cond=condition
    RPAR #condblock;
r_function : FUNC name=func_name LPAR
    args=elements RPAR COLON (sent_mult=
    sentences | sent=sentence)? (RETURN
    LPAR ret_args=elements RPAR)? #
    function;
sentences_func : ((SEMI)? sentence)+ ;
func_name : CHAR_LIT ;
func_call : name=func_name LPAR ele=
    elements RPAR #function_call;
file_name: CHAR_LIT_DOT;
r_list : LSPAR (empty|elements) RSPAR ;
elements : ((COMMA)? literal_expr)* ;
empty : ;
```

Listing 5: EBNF syntax definition for CLAPLEX- Table 2

```
expression : comparison_expr
             | bool_lit;
logic_expr : op=NOT expr=expression #
    notlogic
             | LPAR left=expression RPAR
                 op=andor LPAR right=
                 expression RPAR #andorlog
                 ;
comparison_expr : left=mathematical_expr
     op=comp_oper right=mathematical_expr
      #comp_expr;
var_assign : name=var_name op=
    ASSIGN_OPER (lit_expr=literal_expr|
    math_expr=mathematical_expr) #
    var_assignment;

mathematical_expr : left=
    mathematical_expr op_exp=
    arith_oper_exp right=
    mathematical_expr     #opExpr
                  | left=mathematical_expr
                      op_multdiv=
                    arith_oper_mult_div
                    right=
                    mathematical_expr  #
                    opExpr
                  | left=mathematical_expr
                      op_addsub=
                    arith_oper_add_sub
                    right=
                    mathematical_expr   #
                    opExpr
                  | number=num #defnegnum
                  | variable=var #defvar
                  | boolvar=bool_lit #
                    defbool;

literal_expr :   num
                  | var
                  | bool_lit
                  | str_lit
                  | func_call;

str_lit : QUOTE (CHAR_LIT|WHITESPACE|
    SYMBOL|DOT)* QUOTE ;
bool_lit : BOOL_LIT ;
num : (SUB)? NUM ;
var_name : CHAR_LIT ;
andor : AND|OR ;
log_oper : AND|OR|NOT ;
comp_oper : EQL_EQL|NOT_EQL|GNE|LNE|GEQ|
    LEQ ;
arith_oper_add_sub : ADD|SUB ;
arith_oper_mult_div : MULT|DIVD ;
arith_oper_exp : EXP ;
var : CHAR_LIT ;
```

Listing 6: EBNF syntax definition for CLAPLEX- Table 3

```
IMPORT : ’import’ ;
COMMENT : ’#’ .*? ’/n’ -> skip;
R_TYPE : ’bool’|’str’|’int’|’float’;
EXEC_MODE : ’sequence’|’parallel’|’
    any_order’ ;
BOOL_LIT : ’true’|’false’ ;
PROGRAM : ’program’ ;
LPAR : ’(’ ;
RPAR : ’)’ ;
SEMI : ’;’ ;
COMMA : ’,’ ;
LCPAR : ’{’ ;
RCPAR : ’}’ ;
RSPAR : ’]’ ;
LSPAR : ’[’ ;
QUOTE : ’\’’ | ’"’ ;
FUNC: ’func’ ;
COLON : ’:’ ;
RETURN: ’return’ ;
DO : ’do’ ;
WHILE : ’while’ ;
IF : ’if’ ;
ELSEIF : ’else if’ ;
ELSE : ’else’ ;
FOR : ’for’ ;
IN : ’in’ ;
CHECK : ’check’ ;
CASE : ’case’ ;
AND : ’and’ ;
OR : ’or’ ;
NOT : ’not’ ;
EQL_EQL : ’==’;
NOT_EQL : ’!=’;
GNE : ’>’;
LNE : ’<’;
GEQ :’>=’;
LEQ : ’<=’;
ASSIGN_OPER : ’=’ ;
EXP : ’ˆ’ ;
ADD : ’+’ ;
SUB : ’-’ ;
MULT : ’*’ ;
DIVD : ’/’ ;
NUM : ’1’..’9’+ (’0’..’9’)* |
    ((’0’..’9’)+ ’.’ (’0’..’9’)+) |’0’;
DOT : ’.’ ;
NONZERO : ’1’..’9’ ;
DIGIT : ’0’..’9’ ;
INT : (’0’..’9’)+ ;
CHAR_LIT : (’a’..’z’|’A’..’Z’|’0’..’9’|’
    _’)+ ;
CHAR_LIT_DOT : (’a’..’z’|’A’..’Z
    ’|’0’..’9’|’_’|’.’)+ ;
WHITESPACE : (’ ’|’\t’|’\r\n’|’\n’|’pass
    ’|’#’)+ -> skip;
SYMBOL : ’-’|’_’|’@’|’%’|’&’|’(’|’)
    ’|’:’|’!’|’,’|’?’;
```

References

Browning, B.; Bruce, J.; Bowling, M.; and Veloso, M. M. 2005. STP: Skills, tactics, and plays for multi-robot control in adversarial environments. *Proceedings of the Institution of Mechanical Engineers, Part I: Journal of Systems and Control Engineering*, 219: 33 – 52.

Cashmore, M.; Fox, M.; Long, D.; Magazzeni, D.; Ridder, B.; Carrera, A.; Palomeras, N.; Hurtos, N.; and Carreras, M. 2015. ROSPlan: Planning in the Robot Operating System. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 25, 333–341.

Colledanchise, M.; and Ögren, P. 2017. Behavior Trees in Robotics and AI: An Introduction. *CoRR*, abs/1709.00084.

Colledanchise, M.; and Ögren, P. 2017. How Behavior Trees Modularize Hybrid Control Systems and Generalize Sequential Behavior Compositions, the Subsumption Architecture, and Decision Trees. *IEEE Transactions on Robotics*, 33(2): 372–389.

De Benedictis, R.; Beraldo, G.; Cortellessa, G.; Fracasso, F.; and Cesta, A. 2022. A Transformer-Based Approach for Choosing Actions in Social Robotics. In Cavallo, F.; Cabibihan, J.-J.; Fiorini, L.; Sorrentino, A.; He, H.; Liu, X.; Matsumoto, Y.; and Ge, S. S., eds., *Social Robotics*, 198–207. Springer Nature Switzerland. ISBN 978-3-031-24667-8.

Despouys, O.; and Ingrand, F. F. 1999. Propice-Plan: Toward a Unified Framework for Planning and Execution. In *Proceedings of the 5th European Conference on Planning: Recent Advances in AI Planning*, ECP '99, 278–293. Berlin, Heidelberg: Springer-Verlag. ISBN 3540678662.

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

Goldman, R. P.; Bryce, D.; Pelican, M. J. S.; Musliner, D. J.; and Bae, K. 2016. A Hybrid Architecture for Correct-by-Construction Hybrid Planning and Control. In *NASA Formal Methods*, 388–394. Springer International Publishing. ISBN 978-3-319-40648-0.

Guzmán, C.; Alcázar, V.; Prior, D.; Onaindía, E.; Borrajo, D.; Fdez-Olivares, J.; and Quintero, E. 2012. PELEA: a Domain-Independent Architecture for Planning, Execution and Learning. In *Proceedings of ICAPS'12 Scheduling and Planning Applications woRKshop (SPARK)*, 38–45. Atibaia (Brazil): AAAI Press.

Haigh, K. Z.; and Veloso, M. M. 1997. High-Level Planning and Low-Level Execution: Towards a Complete Robotic Agent. In *Proceedings of the First International Conference on Autonomous Agents*, 363–370. Association for Computing Machinery. ISBN 0897918770.

Ingham, M. D.; Ragno, R. J.; and Williams, B. C. 2001. A Reactive Model-based Programming Language for Robotic Space Explorers. In *In Proc. International Symp. on Artificial Intelligence, Robotics and Automation in Space*.

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.

Macenski, S.; Foote, T.; Gerkey, B.; Lalancette, C.; and Woodall, W. 2022. Robot Operating System 2: Design, architecture, and uses in the wild. *Science Robotics*, 7(66): eabm6074.

Mendoza, J. P.; Veloso, M.; and Simmons, R. 2015. Plan execution monitoring through detection of unmet expectations about action outcomes. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, 3247–3252.

Mericli, C.; Klee, S. D.; Paparian, J.; and Veloso, M. 2014. An Interactive Approach for Situated Task Specification through Verbal Instructions. In *Proceedings of the 2014 International Conference on Autonomous Agents and Multi-Agent Systems*, 1069–1076. International Foundation for Autonomous Agents and Multiagent Systems. ISBN 9781450327381.

OMG, O. M. G. 2021. Decision Model and Notation Specification. https://www.omg.org/spec/DMN/1.3. Accessed: 2023-02-01.

Parr, T. 1992. ANTLR. https://www.antlr.org/. Accessed: 2022-01-15.

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

Patra, S.; Ghallab, M.; Nau, D.; and Traverso, P. 2019b. APE: An Acting and Planning Engine. *Advances in Cognitive Systems*, 7: 175–194.

Patra, S.; Mason, J.; Kumar, A.; Ghallab, M.; Traverso, P.; and Nau, D. 2020. Integrating Acting, Planning and Learning in Hierarchical Operational Models. arXiv:2003.03932.

Pinover, K.; Ferguson, E.; Bindschadler, D.; and Schimmels, K. 2020. The Reference Activity Plan: Collaborative, Agile Planning for NASA's Europa Clipper Mission. In *2020 IEEE Aerospace Conference*, 1–13.

Schaffer, S.; Russino, J.; Wong, V.; Justice, H.; and Gaines, D. 2018. Integrated Planning and Execution for a Self-Reliant Mars Rover. In *International Conference on Automated Planning and Scheduling, Workshop on Integrated Planning, Acting and Execution*.

Simmons, R.; and Apfelbaum, D. 1998. A task description language for robot control. In *Proceedings. 1998 IEEE/RSJ International Conference on Intelligent Robots and Systems. Innovations in Theory, Practice and Applications (Cat. No.98CH36190)*, volume 3, 1931–1937.

Turi, J.; and Bit-Monnot, A. 2022. Guidance of a Refinement-based Acting Engine with a Hierarchical Temporal Planner. In *International Conference on Automated Planning and Scheduling, Workshop on Integrated Planning, Acting and Execution*.

Veloso, M. M.; Pérez, A.; and Carbonell, J. G. 1990. Nonlinear planning with parallel resource allocation. In *Proceedings of the DARPA workshop on innovative approaches to planning, scheduling, and control*, 207—-212. Morgan Kaufmann.

Verma, V.; Estlin, T. A.; Jónsson, A. K.; Pasareanu, C. S.; Simmons, R. G.; and Tso, K. S. 2005. Plan execution interchange language (PLEXIL) for executable plans and command sequences. In *In Proc. International Symp. on Artificial Intelligence, Robotics and Automation in Space*, 1–8.