Expert Systems

WILEY

Expert Systems

ORIGINAL ARTICLE OPEN ACCESS

Towards a No Code Deployment of Social Robotics Use Cases

Alba Gragera[1] | Carmen Díaz-de-Mera[1] | Juan Pedro Bandera[2] | Ángel García-Olaya[1] | Fernando Fernández[1]

[1]Department of Computer Science and Engineering, Universidad Carlos III de Madrid, Av. de la Universidad, Madrid, Spain | [2]Department of Electronic Technology, Universidad de Málaga, ETSI de Telecomunicación, Campus de Teatinos, Málaga, Spain

Correspondence: Alba Gragera (agragera@pa.uc3m.es)

Received: 11 October 2023 | **Revised:** 29 January 2025 | **Accepted:** 8 March 2025

Funding: The authors received no specific funding for this work.

Keywords: automated planning | social autonomous robotics | use case modeling

ABSTRACT

Social Autonomous Robotics aims to deploy robots in scenarios that involve intensive and continuous interaction with humans. To control the behaviour of robotic platforms in such environments, the use of automated planning (AP) within a control architecture has been proposed as an effective mechanism. However, the design of AP models is time-consuming and typically carried out by domain experts and engineers. A significant amount of knowledge must be acquired in order to properly define the use case description by specifying the different tasks performed by the robot. In this paper, we present DEVPLAN, a framework for graphically designing robotic use cases and configuring the platform for the desired execution. DEVPLAN provides an interface that allows domain experts, in collaboration with knowledge engineers, to use state transition diagrams to specify the tasks a robot can perform and define recovery strategies for exogenous events that disrupt normal execution. This graphical design is automatically translated into the standard Planning Domain Definition Language (PDDL). Additionally, to facilitate the integration of the AP model with the robot's control architecture, DEVPLAN includes a module for generating the configuration files required to set up the control system. The proposed framework has been successfully used to design and deploy two different use cases in a real environment in a retirement home.

1 | Introduction

In recent years, there has been a growing interest in robots operating in public spaces. A key area of research focuses on Social Autonomous Robotics (SAR) (Breazeal et al. 2016), which requires these robots to adapt their actions based on the sensor data they collect. They must show flexible capabilities and robust behaviours even in dynamic and constantly changing environments (Ingrand and Ghallab 2017). Automated Planning (AP) (Ghallab et al. 2004) has been used previously to achieve this autonomous behaviour (Bandera et al. 2016; Cashmore et al. 2015; Chen et al. 2016; González et al. 2017; Mohseni-Kabir et al. 2020; Rajan and Py 2012; Tran et al. 2017) by using a problem solver and a control architecture: the problem solver creates a plan of actions to be performed, while the control architecture deals with execution and monitoring, adapting the plan to the changing environment and replanning if necessary.

However, developing autonomous systems for Human-Robot Interaction (HRI) scenarios remains a challenging task (Tapus et al. 2007), particularly in what relates to the time-consuming knowledge engineering process behind the development of these systems (Kambhampati 2007; Bhatnagar et al. 2022). In contrast to more typical uses of knowledge engineering such as classification or diagnosis, AP focuses this process on the creation of models to reason about actions, facts, and states, which will ultimately be processed by specific reasoning engines to synthesise a solution plan (McCluskey et al. 2017).

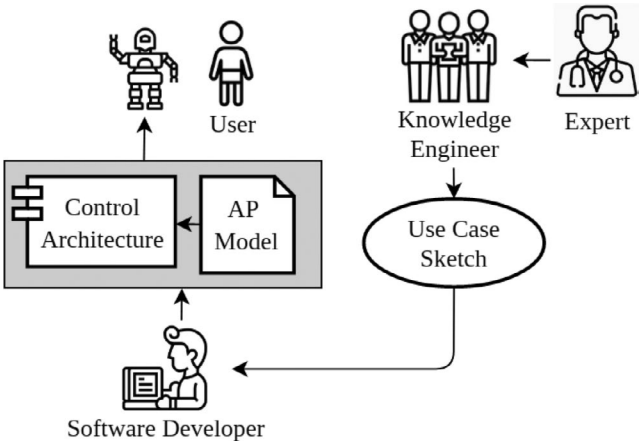


FIGURE 1 | Common development process to build AP-based social robotics use cases.

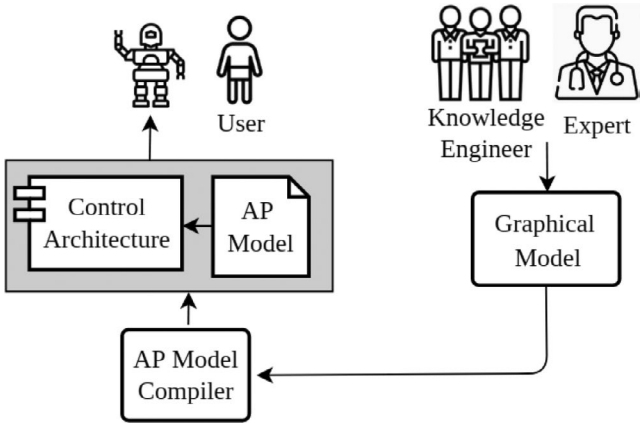


FIGURE 2 | Proposed development process using the graphical interface, where domain experts actively collaborate with knowledge engineers during the modelling process.

This process involves cooperation between domain experts, knowledge engineers, and software developers. The result of this procedure is a formal specification of the planning task, in general using the Planning Domain Description Language (PDDL) (McDermott et al. 1998), the standard action description language adopted by the community. In addition, the generated model has to be integrated into a control architecture configured to convert the sensor data collected into high-level data managed by the planner and the actions proposed by the planner into low-level actions (Figure 1). These challenges are not limited to the AP technique; they extend to any other control technique as well. They act as bottlenecks for developers and pose entry barriers for novice users looking to deploy Social Robotics use cases. This has led to an increasing demand for frameworks that facilitate seamless robot programming for users (Kramer and Scheutz 2007).

In this paper, we present DEVPLAN, our initiative to simplify the creation of SAR use cases, thereby encouraging the participation of experts and knowledge engineers in the development process. With DEVPLAN, they can directly model the desired scenario, observe the sequence of actions it generates, and execute them with the robot, all without requiring software developers to manually code the use case. In this approach, the encoding is handled by a model compiler, as shown in Figure 2. Using AP as the paradigm to describe the use case and a control architecture that implements its model, the contributions of this paper can be summarised as follows:

1. A Use Case representation based on state transition diagrams, where the system's tasks are depicted as sets of workflows.
2. An interface to graphically build such models, representing the expected behaviour of the robot.
3. An automated compilation from the graphical model into the Planning Domain Definition Language (PDDL) formalisation (McDermott et al. 1998), including features to operate in dynamic environments.
4. An interface module to create the configuration files required to configure the control architecture, making it ready to connect with the robotic platform.

All these contributions are brought together under the DEVPLAN framework. Throughout this paper, we demonstrate the validity of the contributions by establishing the following objectives:

- To demonstrate the capability of DEVPLAN in modelling complex social robotics use cases, we illustrate it using a collaborative stacking-blocks game that incorporates all the features of social interaction.
- To assess DEVPLAN with representative users and determine its understandability and intuitiveness for non-experts, we gave computer science students two use case specifications to implement in PDDL, as well as through the proposed interface. Subsequently, they participated in usability tests to evaluate the method's functionality and complexity.
- To ensure the correctness and executability of the generated models, we developed two use cases drawn from the needs identified in a retirement home. These use cases were created from scratch in DEVPLAN and then deployed on a robot for interaction with older adults in a real environment.

The accomplishment of these objectives demonstrates the feasibility of our work, representing the initial steps towards a no-code development approach in HRI.

The paper is organised as follows. Section 2 introduces the concepts of AP and control architectures needed for this work. Section 3 details the knowledge elicitation to build AP tasks. Section 4 explains how such knowledge is modelled in DEVPLAN though the concept of *options*. Once the model is in the framework, Section 5 contains how DEVPLAN compiles the graphical model into the corresponding PDDL formalisation. After that, Section 6 details how such generated model can be integrated into a real robotic platform, without the need of further code. Section 7 involves the evaluation performed according to the objectives, divided in usability tests of DEVPLAN with untrained users and the field trials carried out in a retirement home. Finally, Section 8 compares the proposed approach with a previous one based on state machines, and Sections 9 and 10 show the related work and the main conclusions of the work.

2 | Background

AP deliberative systems must cope with uncertainty, especially in real-world deployments, and depend on a control architecture to collect external information for use in the deliberation process. In this section, we provide background information on AP, common approaches to managing uncertainty, and examples of control architectures.

2.1 | Automated Planning Paradigms

An AP task consists in finding a set of actions, called a plan, which applied to a given initial state reaches a state where the goals are achieved. We use the first-order (lifted) planning formalism, where a classical planning task is a pair $\Pi = (D, I)$, where D is the planning domain, and I defines a problem instance. A planning domain is a tuple $D = \langle \mathscr{H}, \mathcal{P}, \mathcal{A} \rangle$; where $\mathscr{H}$ is a type hierarchy; $\mathcal{P}$ is a set of predicates defined by their names and the types of its arguments; and $\mathcal{A}$ is a set of *action schemas*. If $p(t) \in \mathcal{P}$ is an n-ary predicate and $t = t_1, \dots, t_n$ are either typed constants or typed free variables, then $p(t)$ is an *atom*. An atom is *grounded* if its arguments do not contain free variables. Action schemas $a \in \mathcal{A}$ are tuples $a = \langle \text{par}(a), \text{pre}(a), \text{eff}(a) \rangle$, defining the action parameters (a finite set of free variables) $\text{par}(a)$; the preconditions $\text{pre}(a)$; and effects $\text{eff}(a)\rangle$. $\text{pre}(a)$ is a set of atoms representing what must be true in a state to apply the action. $\text{eff}(a)$ represent the changes produced in a state by the application of the action (added and deleted atoms).

Additionally, we also consider a set of partial states $\mathcal{P}_s$ composed by sets of lifted atoms that are expected to be grounded with the corresponding objects during the execution of the system. A problem instance is a tuple $I = \langle \mathcal{O}, \mathcal{I}, \mathcal{G} \rangle$, where $\mathcal{O}$ is a set of typed constants representing problem-specific objects; $\mathcal{I}$ is the set of ground atoms in the initial state; and finally, $\mathcal{G}$ is the set of ground atoms defining the goals.

Grounded actions a are obtained from action schemas a by substituting the free variables in the parameters of the action schema by constants in $\mathcal{O}$. A grounded action a is applicable in an state s if $\text{pre}(a) \subseteq s$. When a grounded action is applied to s we obtain a successor state s', defined as $s' = \{s \setminus \text{del}(a)\} \cup \text{add}(a)$. A plan π is a sequence of grounded actions $a_1, \dots, a_n$ such that each a_i is applicable to the state s_{i-1} generated by applying $a_1, \dots, a_{i-1}$ to $\mathcal{I}$; a_1 is applicable in $\mathcal{I}$; and the consecutive application of all actions in the plan generates a state s_n containing the goals $\mathcal{G} \subseteq s_n$.

2.2 | Dealing With Uncertainty in AP

While there are various paradigms within the field of Automated Planning (AP), the definition provided above assumes that action outcomes are known and the environment is deterministic, thus requiring no observability. However, the real world is non-deterministic, with actions prone to failure and external agents capable of altering the environment unexpectedly. Human-Robot interactive tasks are good examples of such scenarios, where accurately predicting human actions is challenging. Consequently, observations become essential for validating the true state of the world. In this way, planning with sensing has been studied in the literature under Contingent Planning, where a real execution is a combination of actions and observations (Bonet and Geffner 2013). Solving the *Belief Tracking for Planning* problem means determining the possible observations that may result after the execution, identifying applicable actions and assessing goal achievement, ultimately resulting in a complete policy. Dealing with the uncertainty of the world has also been addressed through Probabilistic Planning models, typically formulated using Markov Decision Processes (MDP) (Puterman 1994) or languages like PPDDL (Younes and Littman 2004) or RDDL (Sanner 2011), an extension of PDDL designed to express probabilistic planning domains that handle uncertainty with probabilistic action effects. It permits a management of possible unforeseeable events, but requires accurate probabilistic specifications to be established beforehand. Fully Observable Non-Deterministic planning (FOND) (Rintanen 2004) is commonly used to represent non-deterministic domains where each action has multiple possible outcomes that must be explicitly stated in the PDDL domain model. For example, the action of picking up a block may have two possible outcomes: successfully picking it up or accidentally dropping it. In this work, we not only represent the possible outcomes of actions but also account for exogenous events that can occur at any time. For instance, if the robot is greeting a child, the child leaving the room is not a possible outcome of the greeting action but rather an exogenous event that interrupts it. Epistemic Planning (Bolander and Andersen 2011) provides tools for reasoning about agents' knowledge and beliefs, but is not well suited for our objectives. DEVPLAN focuses on modelling observable actions and exogenous events that disrupt the nominal behaviour of a single robot system. Since our approach does not involve reasoning about internal knowledge states or multiple agents, the added complexity of epistemic planning is unnecessary.

A popular way to tackle the inherent uncertainty of the world is to rely on deterministic planning and replan upon unexpected situations (Geffner and Bonet 2013), which provides efficiency in real-time tasks. In this case, the model design assumes a nominal behaviour: a desired flow of actions with enough likelihood of being executed without interruptions (García-Olaya et al. 2019). If an unexpected state is observed during the execution of the plan and the remaining plan cannot be applied, a replanning process can be performed to detect failures or even opportunities (Yoon et al. 2007). Although it may seem simplistic, this planning and replanning approach has been used in many real applications (Bandera et al. 2016; Cashmore et al. 2015; Chen et al. 2016; González et al. 2017; Mohseni-Kabir et al. 2020; Rajan and Py 2012; Tran et al. 2017; McGann et al. 2008). Although other models typically focus on computing policies with belief states that may never be reached during execution (Muise et al. 2014), the present work takes advantage of this approach, relying on the speed and efficiency of deterministic planners.

2.3 | Replanning Strategies

If uncertainty is not inherently incorporated into the domain model, discrepancies may occur between the expected state value and its observed value, rendering the current plan invalid.

In such cases, two primary options are available: plan repair or replanning from scratch (Fox et al. 2006). Plan repair involves adapting the plan to the new situation based on environmental information while minimising alterations to the original plan. In contrast, replanning entails generating an entirely new plan, without considering the previous one. Another alternative is based on the concept of *planning with reuse* (Borrajo and Veloso 2012), assuming that similar past solutions can guide the search for a new plan. However, it is important to note that current plan repair/replanning techniques operate under the assumption that "the plan that has already been executed cannot be retracted, so we can always consider the problem as though the current state were the initial state and the remainder of the unexecuted plan were the whole of the original plan" (Borrajo and Veloso 2012).

However, there are cases where simply setting the current state as the initial state may not suffice, and the execution needs to be recovered from a point earlier than where the interruption occurred. For instance, if readers of this paper are interrupted at this point and return to it tomorrow, they may choose to restart not precisely at this paragraph but maybe at the beginning of the section, just to place themselves again in the context of the paper. In such situations, when restoring the flow, the new initial state may not be the current state but rather a previous one. In this work, we introduce a novel AP compilation to address this challenge.

2.4 | Monitoring and Execution Architectures

A planning approach with replanning upon failure requires monitoring and execution control architectures. Examples include PELEA (Celorrio et al. 2008) and ROSPLAN (Cashmore et al. 2015), both using Classical Planning (Ghallab et al. 2004). They typically involve a planner and a formal planning model to generate a sequence of actions to be performed while verifying the correct execution of the initial plan. Each action is sent to the robotic platform, assuming that no interruptions will occur during execution. To confirm whether the plan is progressing as expected, environmental information is obtained from sensors. If discrepancies arise between the expected state value and its observed value, the current plan may no longer be valid, triggering a replanning process to replace it with a new plan to address the current situation.

MLARAS (Multi-layered Architecture for Autonomous Systems), a similar AP architecture to the ones mentioned, was developed in the context of the NAOTHERAPIST project (González et al. 2017). This architecture integrates planning, execution, monitoring, replanning and learning in different layers of abstraction. Normally, the high-level layer is for deliberation, and a low-level layer is for information that the robot can directly work with. To perform high-level deliberation and translation between layers, MLARAS uses PELEA as a sub-architecture. The general process is shown in Figure 3, where the planner returns the plan $\pi = \{a_1, a_2, \ldots, a_n\}$. Each $a_i \in \pi$ is translated into low-level commands that the robot can execute, which are sent to the robotic platform. The information from the robot sensors is translated into high-level predicates for monitoring purposes. Although initially implemented for

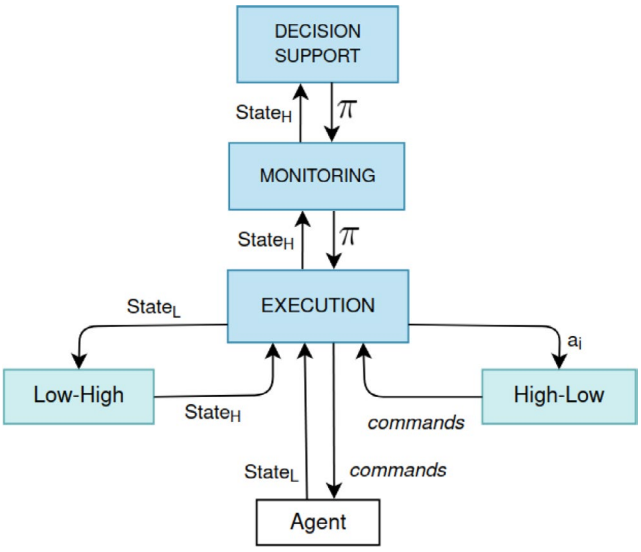


FIGURE 3 | Deliberation layer of the MLARAS. The execution module translates data between high and low levels and monitors plan progress.

a specific project, MLARAS was designed as a generic architecture, which makes it easily adaptable for integration into any robot and in any use case through a minimal configuration process. This flexibility led us to choose this architecture for integrating the use case definitions in the retirement home robot. However, before proceeding, we needed to define the use case models. The next section describes our approach to use case formalisation.

3 | Use Case Design Though Classical Automated Planning Concepts

In this section, we show the AP knowledge that needs to be specified and how it is used to construct the workflows that define the use case. To provide a clearer illustration, we will use a running example: a social robotics use case involving stacking blocks in a specific order to encourage collaboration among participants.

3.1 | Domain Elicitation

The domain represents the information that can be involved in the use case and how it changes as data are added, modified, or deleted by actions. According to the aforementioned, the user has to specify a planning domain represented by the tuple $D = \langle \mathcal{H}, \mathcal{P}, \mathcal{A} \rangle$. Additionally, we consider a set of partial states $\mathcal{P}_s$ that are expected to be transited during the execution of the use case.

3.1.1 | Type Hierarchy ($\mathcal{H}$)

It defines the kind of objects involved in the execution (places, people, items, etc.), that can be related with other object types. Object-type hierarchies can be created to define generalisations and specialisations. For example, in a use case where robots and

children play together, the child and robot types can both belong to the player super-type.

3.1.2 | Predicates ($\mathcal{P}$)

They are mostly used to create relations between objects useful during the execution, or to describe features or situations that involve them, the environment or internal control knowledge. Using predicates logic, users are asked to specify the predicate name and the free-typed variables $t = t_1, \ldots, t_n$ they involve. A declaration like (holding ?b—block ?p—player) represents a lifted atom $p(t) \in \mathcal{P}$ which can be grounded with objects of type block and player, meaning that a player is holding a certain block: (holding block02 player01). Generally speaking, we can find different categories of predicates

- Static/Dynamic: The former represents persistent information that does not change as a result of action executions, such as the locations of rooms in a building. Dynamic predicates, on the other hand, can be added, removed, or modified by the application of actions or external events.
- Internal/Sensed: Internal predicates are used to represent internal data explicitly calculated and updated within the system, such as determining which player's turn it is to move a block. Sensed predicates, on the other hand, represent characteristics of the real world and can only be updated through sensing. For example, they can represent two blocks stacked one upon the other or external events interrupting the use case, such as a child suddenly leaving the room. It is important to consider these types of sensed events to build robust models.

The user must distinguish between static/dynamic and sensed/internal predicates during predicate specification in DEVPLAN, depending on the type of information being represented. This distinction is crucial when recovering from errors, as it helps differentiate between information collected from the environment and that which is internally managed by the system.

3.1.3 | Partial States ($\mathcal{P}_s$)

Interpreted as a conjunctive formula, a partial state is specified as a subset of lifted atoms grounded with the current true information, representing the facts that the agent must consider. The following example illustrates a lifted partial state in which both the child and the robot are ready to start the game. This state will be instantiated with the current child participating in the defined game. Other components of the full state, such as the location of objects, may be omitted since they are not relevant to the current reasoning step.

```
(detected-child ?c - child)
(training-area ?l - location)
(game ?g - game)
(robot-at ?l - location)
(child-at ?l - location)
(robot-idle)
```

3.1.4 | Actions ($\mathcal{A}$)

Following the scheme $a = \langle \text{par}(a), \text{pre}(a), \text{eff}(a) \rangle$, a simple way to elicit them is to ask the expert what characteristics the scenario must have to perform an action ($\text{pre}(a)$) and how the state changes after its execution ($\text{eff}(a)$). The preconditions must match a defined partial state $\text{pre}(a) \subseteq ps, ps \in \mathcal{P}_s$, while the effects are specified in the description of the action, along with its name. Parameters are gathered from the lifted typed variables in both precondition and effects.

Figure 4 illustrates the action that starts the game. The first box represents a state where the child has been detected. The child is denoted by the lifted variable c. Both the child and the robot are located in the same training area, represented by l. The robot is idle, and a game (g) has been established. The start-game action, shown in the diagram, can be executed to perform changes in the current state. Specifically, it sets the robot to a *training* mode, removing the old state and initiating the game (g). It also introduces a proposition to indicate that the current phase is playing. As this representation captures a partial state, additional information (such as the current locations of blocks or whether the child has been greeted) remains part of the world but is not relevant for the current step. Therefore, it can be omitted from this state representation and included only when necessary.

3.2 | Problem Specification

A planning problem provides relevant information for the current use case that the domain must consider: the initial state and

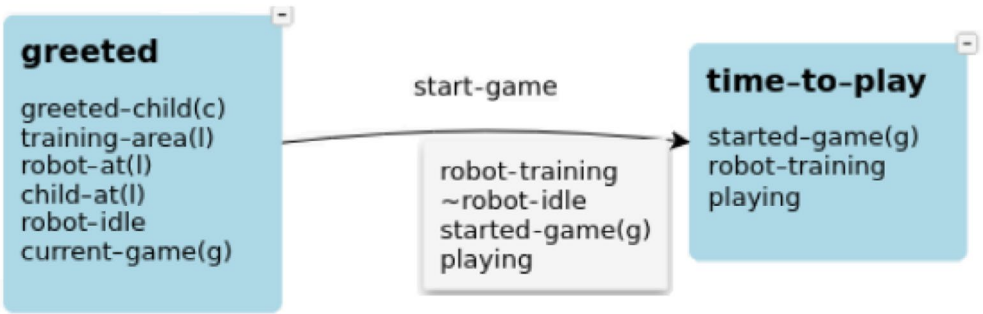


FIGURE 4 | Representation of two partial states (blue boxes), containing the predicates that must be satisfied to execute the action (arrow).

the goals to achieve. For each type defined in the domain, users are required to create associated objects and instantiate predicates based on the current use case. For example, the following information represents a problem in which a child is engaged in a single-block game with a robot, featuring four different blocks labelled A, B, C, and D. In this scenario, blocks A and C are on the table, while blocks B and D are respectively on top of them.

```
;Objects
child - child01
playground - location
blocks - game
A B C D - block

;Initial state
(training-area playground)
(robot-at playground)
(current-game blocks)
(on-table A)
(on B A) (on D C) (on-table C)
```

Goals may consist of a single predicate or a set of facts that the agent must achieve. A goal state is a partial state in which all the goals are true. The entire decision-making process is directed towards achieving these goals during execution. In the context of the defined problem, an example of a goal is to complete the game with a specific arrangement of blocks, such as inverting the two stacks of blocks: (finish-game blocks) (on A B) (on C D).

The specification of these components forms an agent-based model that, when integrated into a cognitive architecture, enables the system to reason about the behaviour it should exhibit in order to function autonomously.

4 | Graphical Modelling

To simplify the process of creating Automated Planning (AP) models based on the concepts mentioned above, DEVPLAN introduces workflows as a visual representation of the knowledge of the use case. Unlike other approaches, such as state machines, which require a complete specification of states and all possible transitions, our method recognises that experts may not always have full knowledge of the entire sequence of actions needed to solve a problem.

Additionally, external factors can disrupt the normal sequence of actions, especially in uncontrolled environments. With our approach, it is not necessary to fully specify states and transitions. Instead, partial definitions with some facts are sufficient for reasoning about the state and are easier to define. We propose using disconnected graphs to represent each executable system task in isolation, allowing the planner to select the most promising sequence of actions to achieve the goals.

Figure 5 shows the stacking blocks game example as modelled in DEVPLAN, with the different tasks the robot can perform depicted in the graphical area. The blue boxes represent partial states, consisting of the facts that the scenario must satisfy to execute the subsequent action, as shown in the expanded states in Figure 4. Each fact is represented by its predicate symbol, along with the associated lifted-typed variables in brackets. If

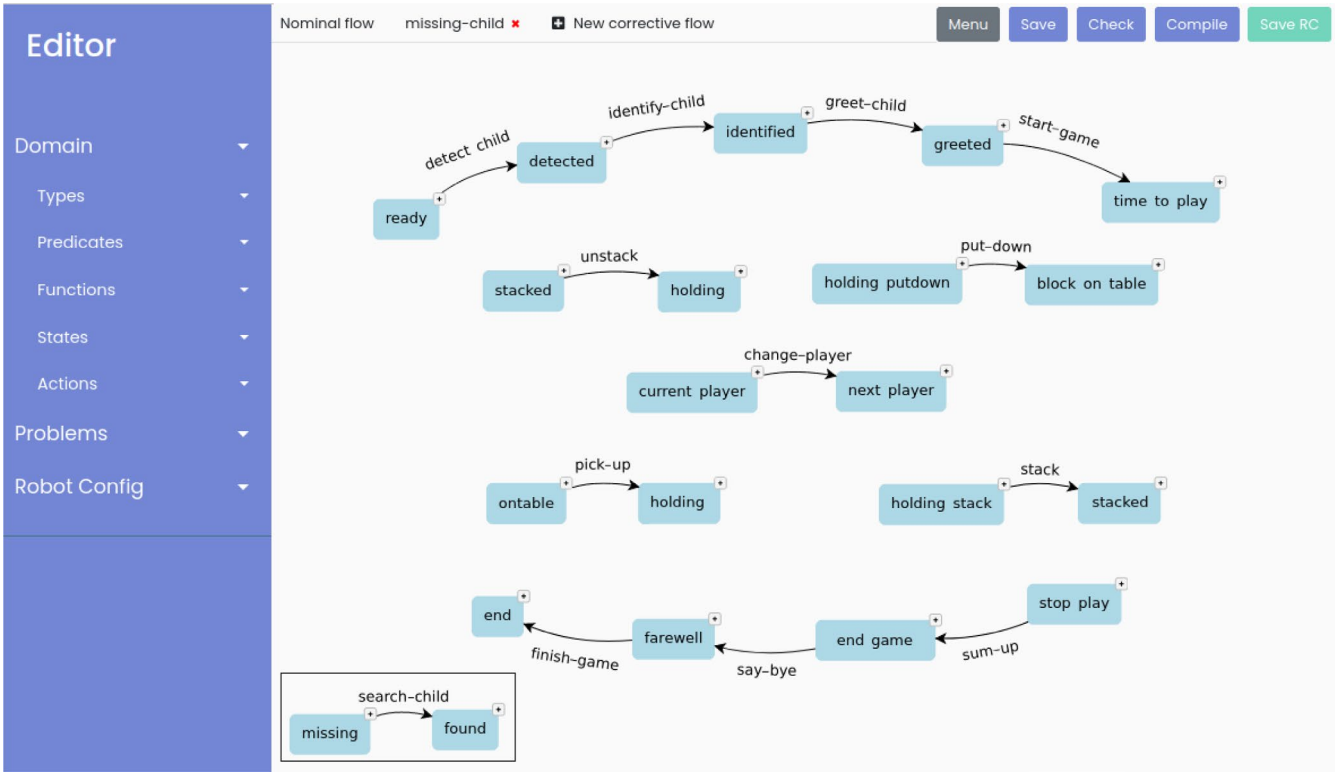


FIGURE 5 | Stacking blocks social robotics use case modelled in DEVPLAN. The left-side menu enables users to define the domain and problem according to AP concepts, which are displayed in real time in the white area. Blue boxes contain the relevant information each state must hold to execute the action. Actions are the edges connecting states.

an atom is preceded by the symbol ∼, it indicates that the predicate must not be true at that point to consider the state.

Actions are represented as edges connecting states, illustrating the expected transitions between states. They specify the information that is added or removed from the state from which they originate. To assist in visualisation, action effects can be viewed in a tool-tip (Figure 4).

4.1 | Nominal *Options*

The graph is designed to represent different stages: the session introduction (connected states in the upper part of the figure), playtime (disconnected graphs in the middle), and the farewell (lower part of the figure). It illustrates the 'nominal behaviour,' which includes the desired set of actions and states for the AP system. During the playtime stage, various alternatives can be pursued by both the child and the robot, such as picking up or unstacking a block. However, it is uncertain which block will be moved first, as this depends on their initial locations and the child's decisions. This uncertainty is why the playtime stage is modelled using disconnected graphs. Then, this use case would be valid for a problem involving a tower of blocks ABC, with the goal of placing all the blocks on the table, leading the planner to apply the actions *unstack* and *put down*. It could also apply to a scenario where the goal is to rearrange the blocks to form the tower CBA, where the planner will also include the actions *stack* and *pick up*.

Then, this nominal behaviour is depicted as a directed graph composed by partial states and actions, $\langle \mathcal{P}_s, \mathcal{A} \rangle$, in such a way that applying $a \in \mathcal{A}$ in $ps \in \mathcal{P}_s$ results in ps'. It can be formalised using the concept of *options* (Sutton et al. 1999):

Definition 1. An **option** is a tuple $o = \langle \mathcal{I}, \pi_o, \beta \rangle$ where $\mathcal{I} \subseteq \mathcal{P}_s$ is an initiation set, π_o a partial policy and β the termination condition.

An option is considered applicable in a state ps_t only if $ps_t \in \mathcal{I}$. Once the option is chosen, the next action $a_t \in \pi_o$ is selected, resulting in a transition to state ps_{t+1}. If the action $a_{t+1} \in \pi_o$ is not applicable in such state, it reaches the termination condition, allowing for the selection of any other option. Options can be partially executed and resumed from any point, so if there is a desire to return to an option, it does not have to be started from the beginning. This assumes that all states where an option might continue are also states where the option can be initiated (Sutton et al. 1999).

Options are graphically specified by depicting the causal links between actions $a_i \xrightarrow{p} a_n$, in which p is both an effect of a_i and a precondition of a_n, establishing an order constraint $a_i < a_n$. However, neither transitions between options nor termination conditions are explicitly stated, and order restrictions are only used to easily depict and understand the current use case, they are not forced during the planning process. Users only need to outline them as a reasonable framework for the use case tasks, even if they are incomplete. The problem solver will search for the correct order of options to achieve the goal, filling in the gaps with other options.

4.2 | Handling Uncertainty: Recovery Options and Checkpoints

Nominal options model the desired behaviour of the system, but exogenous events may occur and interrupt it. Therefore, it is essential to model *recovery options*: actions capable of handling the current situation and restoring the normal flow of the use case. An example of this is searching for the child when they are no longer present. Without such recovery options, the system may become stuck with no means of restoring itself.

Recovery options are modelled separately in DEVPLAN using minor graphs, which can be added or switched in the upper tabs. These graphs indicate the actions needed to resolve unexpected situations. They are not only used to model unexpected events but also to address stochastic actions. In cases where an action has multiple possible outcomes, one of them is included in the nominal behaviour, while the others are modelled as exogenous events. For example, if all the robot's actions have the probabilistic outcome of a low battery level, this outcome would be managed with a recovery option, as the battery level could become low in almost any situation.

As discussed in Section 2.3, it may not always be desirable to replan from the current state, but rather to resume the execution from an earlier point. If partial states are subsets of lifted atoms, users can define a subset of these partial states as *checkpoints* within the nominal behaviour. These checkpoints serve as states from which the execution can be recovered after a failure. Recovery will occur from the last partial state visited. For instance, in the use case depicted in Figure 5, a corrective option is available for situations where the child is lost. In this scenario, the states *time to play* and *stop play* are marked as checkpoints. Thus, if the child leaves the room, the system can either restart the game or summarise to conclude the session, depending on the stage at which the interruption occurred. It is important to note that we refer to exogenous events as those that are not part of the nominal behaviour but are, to some extent, anticipated. If a failure is truly unexpected, it will not be modelled, and the system will be unable to recover from it.

Then, we propose DEVPLAN as a means to describe Automated Planning (AP) use cases using *options*, *recovery options* and *checkpoints*. The framework displays this information in real time within the visual interface and also allows users to define problems associated with the represented domain, specifying the initial state (grounded atoms) of the world and the goals to be achieved. Although DEVPLAN is primarily developed for modelling social robotics use cases, its versatility allows it to be applied to a wide range of Automated Planning (AP) applications. It is inherently designed to handle increasing complexity through the use of partial state definitions and modular task modelling, enabling the system to scale effectively as the number of interactions expands. Even multi-robot systems can be represented in DEVPLAN by modelling different robots as objects, with the control architecture responsible for sending information to the appropriate robot. An example of this approach can be found in (González et al. 2020).

5 | AP Compilation

DEVPLAN simplifies the process of gathering the requirements for the Automated Planning (AP) system. However, after this stage, the code development remains a tedious and complex task, especially when dealing with challenges related to defining deterministic actions to model stochastic domains while achieving natural interaction. These challenges have hindered the adoption of AP as a paradigm in this field. In this section, we present a translation process from the graphical model created in the editor to a high-level declarative language. The model is initially saved in an XML format, capturing all the information about states, actions, as well as initial and goal states. We chose the XML format because it allows flexibility for various compilations into the desired target language based on specific requirements. For this work, we consider a translation into the standard PDDL 2.1 formalisation (Fox and Long 2003). Below, we present the implemented algorithms for converting the model, structured around options, recovery options, and checkpoints, into PDDL.

5.1 | Nominal Options

Options of the nominal behaviour are described by means of sequential connections of states and loops, translated by DEVPLAN as follows.

5.1.1 | Sequential Connections

States in sequential connections contain facts that the environment must hold to execute the action, essentially defining the action's precondition. Effects are derived from the action definition (as shown in Figure 4) and parameters are obtained from the lifted variables included in preconditions and effects. The nominal behaviour is executed as long as no information compromising the expected state is received. Therefore, every predicate related to an unexpected event is included as a negated precondition of each nominal action. This ensures that these actions will not be executed in case of interruption. The code in Figure 6 represents the translation of the action depicted in Figure 4, where the (not (missing-child ?c)) condition comes from the unexpected event (missing-child ?c). If the (missing-child ?c) fact were present, it would be addressed by the search-child action.

```
(:action start-game
  :parameters(?c-child ?l-location ?g-game)
  :precondition (and (greeted-child ?c)
                (training-area ?l)
                (robot-at ?l) (child-at ?l)
                (current-game ?g)
                (robot-idle)
                (not (missing-child ?c)))
   :effect (and (not (robot-idle))
                (robot-training)
                (started-game ?g)
                (playing))
 )
```

FIGURE 6 | Start-game action PDDL formalisation.

5.1.2 | Loops

DeVPlan allows modeling loop actions in two different ways.

- In *for-like* loops, an option includes the repetition of a sequence of actions a specified number of times. These loops are defined using two actions leaving from the same partial state $ps \in \mathcal{P}_s$. One of the actions (a_{in}) enables to keep inside the loop, while the other one (a_{out}) incorporates an exit condition to leave the loop when reached. The exit condition involves a counter, which is incremented or decreased during the cycles until it reaches the exit condition.
- In *while-like* loops, a set of options is repeated as long as a condition remains true. For example, the start-game action in Figure 6 introduces the fact playing, and the action sum-up (depicted in Figure 5) removes it. Consequently, actions related to the game, such as picking up a block or stacking it, can only be executed while this fact is present.

The procedure for generating nominal flow actions is shown in Algorithm 1, which receives all the model information as input. To begin, the function getExogenousFacts receives the set of recovery options and collects all the exogenous facts that can occur in the use case (e.g., the (missing-child ?c) atom). The main for loop (spanning lines 1 to 14) iterates through each action, considering the previous state (getPreviousState). The method isEndOfLoop detects whether the current action marks the end of a *for-like* loop. If it does, the precondition state includes both the numerical condition to exit the loop and to stay in. The method clearExitConditions maintains only the exit condition in the precondition, removing the other possibility. If the action is not part

ALGORITHM 1 | translateNominalAction.

Input: $\mathcal{A}$: the set of actions $\mathcal{P}_s$: the set of partial states $\mathcal{R}_o$: the set of recovery options

Output: A': a set of PDDL actions

1: $A' \leftarrow \emptyset$
2: $E \leftarrow$ getExogenousFacts $(\mathcal{R}_o)$
3: **for** $a_i \in \mathcal{A}$ **do**
4: $\quad s_j \leftarrow$ getPreviousState $(\mathcal{P}_s, a_i)$
5: $\quad a_{\text{out}} \leftarrow$ isEndOfLoop (a_i)
6: $\quad$ **if** a_{out} **then**
7: $\quad\quad \text{pre}(a'_i) \leftarrow s_j \setminus$ clearExitCondition (s_j)
8: $\quad$ **else**
9: $\quad\quad \text{pre}(a'_i) \leftarrow s_j$
10: $\quad$ **end if**
11: $\quad \text{pre}(a'_i) \leftarrow \text{pre}(a'_i) \cup \{\neg p(t) | p(t) \in E\}$
12: $\quad \text{eff}(a'_i) \leftarrow \text{eff}(a_i)$
13: $\quad \text{par}(a'_i) \leftarrow \cup\{\{t\} | p(t) \in \text{pre}(a'_i) \cup \text{eff}(a'_i)\}$
14: $\quad A' \leftarrow A' \cup a'_i$
15: **end for**
16: **return** A'

of a loop, all predicates from the state are added as preconditions (line 8). In both cases, exogenous predicates collected from the recovery options are added as negated preconditions (line 10) since nominal flow actions cannot be executed in the presence of external events. The effects are derived from the action definition, and the parameters are obtained from the objects present in both the preconditions and the effects.

The initial and goal state are defined in DEVPLAN by instantiating atoms with defined objects, so the translation to PDDL formalisation is direct.

5.2 | Recovery Options and Checkpoints

Recovery options for managing exogenous events are generated similarly to the nominal flow options. However, their effects also include the artificial proposition (backwards action) to control the activation of checkpoint recovery. Figure 7 displays the resulting PDDL for the recovery option that accounts for the possibility of the child leaving at any time.

Checkpoint recovery is managed by what we refer to as *backward actions* (Figure 8). The compiler automatically generates one of these actions for each checkpoint defined, removing all intermediate effects added between two different checkpoints. This action forces the restart of the nominal flow from the desired point. Otherwise, the current state would still include all these facts, going back to the exit point when the exogenous event happened. However, it is important to note that predicates marked as persistent or sensed will not be removed; we cannot "restore" information that should be sensed from the environment.

Under normal circumstances, the standard high-level response to an exogenous event during nominal behaviour would involve halting the execution, applying the corrective action(s) (as shown in Figure 7) to resolve the situation, and incorporating the appropriate backward action (Figure 8) to eliminate all intermediate effects, thus restoring the nominal behaviour from the last visited checkpoint. Backward actions are included only if checkpoints are defined; otherwise, the nominal behaviour will be restored from the current state.

```
(:action search-child
  :parameters (?c - child)
  :precondition (missing-child ?c)
  :effect (and (detected_child ?c)
               (not (missing-child ?c))
               (backwards-action))
 )
```

FIGURE 7 | Recovery action.

```
(:action restore-from-checkpoint
  :parameters (?g - game)
  :precondition (and (backwards_action)
                (= (checkpoint) 1))
  :effect  (and (not (started-game ?e))
           (not (backwards_action)))
 )
```

FIGURE 8 | PDDL formalisation for backwards action.

ALGORITHM 2 | createBackwardsAction.

Input: $\mathcal{A}$: the set of actions $\mathcal{P}_s$: the set of partial states

Output: B: the set of PDDL backwards actions

1: $k \leftarrow 0$
2: $b_k \leftarrow \emptyset$
3: **for** $a_i \in A$ **do**
4: $s_i \leftarrow$ getPreviousState $(\mathcal{P}_s, a_i)$
5: checkpoint $\leftarrow$ isCheckpoint (s_i).
6: **for** $p(t)$ in s_i **do**
7: **if** $p(t) \neq$ sensed & $p(t) \neq$ permanent **then**
8: $\text{eff}(b_k) \leftarrow \text{eff}(b_k) \cup \neg p(t)$
9: **end if**
10: **end for**
11: **if** checkpoint **then**
12: $\text{pre}(b_k) \leftarrow$ currentCheckpoint(k)
13: $\text{par}(b_k) \leftarrow \cup\{\{t\} \mid p(t) \in \mathit{eff}(b_k)\}$
14: $B \leftarrow B \cup b_k$
15: $k \leftarrow k + 1$
16: $b_k \leftarrow \emptyset$
17: **end if**
18: **end for**
19: **return** B

The process of generating backward actions is outlined in Algorithm 2, which takes the set of actions and states involved in the model as input. Initially, it creates an empty template for a backward action (line 2). Subsequently, it iterates over the actions, saving all the negated effects of the actions, excluding those that are sensed or permanent. When a checkpoint state is found, the backward action is increased with the collected information. The currentCheckpoint method (line 12) introduces the current checkpoint number and the flag (backwards action) as preconditions of the action, ensuring that the operator is only activated when necessary. Line 14 marks the completion of the current backward action, and a new template is created in line 16, which will be filled if more checkpoints are identified. Finally, the set B of backward actions is returned.

6 | Integration in the Robotic Platform

This section provides detailed insights into the no-code deployment process after the PDDL formal models are generated by DEVPLAN, including their integration into a real robotic platform. We will begin by explaining the deliberative architecture in which the model is integrated, responsible for controlling and monitoring the robot's progress. Then, we will explain the connections between the layers of this architecture, where high-level instructions are translated into the low-level commands interpreted by the robot.

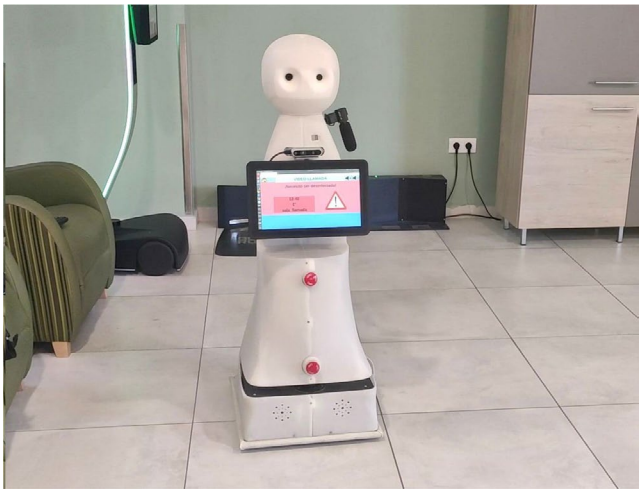


FIGURE 9 | CLARC robot displayed at the retirement home.

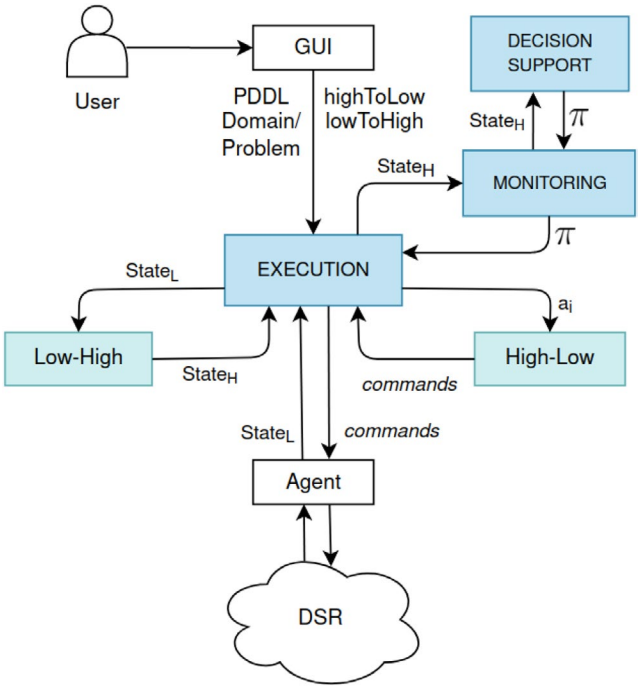


FIGURE 10 | Architecture of the deliberative module of the CLARC robot.

6.1 | Embedding in an AP Control Architecture

The current work uses MLARAS, a planning and replanning architecture. It was implemented on a social robot developed for geriatric assessment in the context of the CLARC project (Martínez et al. 2018) (Figure 9). This robot integrates the ROBOCOMP framework (Manso et al. 2010), a component-oriented architecture whose main aim is to ease the development of robotic frameworks. This middleware automates the communication among different components through TCP/IP using Ice (Internet Communication Engine) interfaces. Moreover, the CLARC robot incorporates the CORTEX cognitive architecture to control its behaviour. In the CORTEX architecture (Bustos et al. 2019), a set of software components communicate with each other through a shared common world view, where all the information the robot has about its context –internal and external –is included. This world representation is coded as an oriented graph structure named Deep State Representation (DSR) (Bustos et al. 2015), that stores symbolic and geometric information. That DSR, the components in charge of different tasks, and the *agents* that link these components to the DSR, conform the CORTEX cognitive architecture.

The planning architecture MLARAS is integrated as a component of this framework, together with its *agent*, that reads and modifies the DSR. The agent will receive the low-level actions of MLARAS and write them into the DSR, so that the other components of the robot will read the information and react accordingly, executing the instructions. In the same way, the agent will read the low-level information written in the DSR and communicate it to MLARAS by means of the low-level variables. Hence, to integrate MLARAS in CORTEX, the main task is to define the interfaces between MLARAS and its agent (from MLARAS to the agent and vice-versa).

The full architecture of MLARAS implemented in the CLARC robot is shown in Figure 10. A high-level layer is used to define the use case by means of the graphical user interface as proposed in this paper. From the interface, the PDDL domain and problem definition are obtained, as well as the translation from high to low level and vice-versa. This output is received by the medium layer, which is the deliberation layer of the architecture. The Execution component centralises all operations, translating between high and low levels. The Execution component communicates directly with the ROBOCOMP agent, getting access to the world view of the DSR, as discussed.

6.2 | From High to Low Level and Vice-Versa

When using deliberative architectures such as the aforementioned, knowledge is generally divided into two levels of abstraction. The external information that comes from the sensors of the robot is low-level and should be translated into a higher level of abstraction so that the planner system integrated in the architecture can reason with it. In our case, the low-level information would be translated into high-level PDDL predicates, which represent the state of the system. In the other direction, high-level actions defined with PDDL should be translated into the low-level instructions that a robot can immediately execute, which are considered low-level actions. For example, in a robot with speakers and a screen, the high-level action *say* could be decomposed into playing a sound and then saying the actual speech while showing subtitles on the screen.

These translations are generally hard-coded directly into the architecture by technical experts, which makes them difficult to change. To alleviate this problem, a declarative language was proposed in (González et al. 2018). That eases the translation, but it is still necessary to understand the Extended Backus–Naur Form (EBNF) description of the grammar for both translations. For instance, a high level action such as say can be converted into different low level commands, like play a sound, show subtitles on a screen, and say que speech:

```
High: SAY(speech)
Lows: play_sound()
      show_subtitles(?speech)
      say(?speech)
```

On the other side, when an event is received by the robot, for example, a scheduled call has been cancelled, that information has to be added to the current state as a PDDL predicate, to reason about it:

```
If: $call_cancelled is true
    add(call_cancelled $patient, state_pddl)
```

To make the translation from high to low level and vice-versa even easier, it has been integrated in DeVPlan, so that when the high-level behaviour of the use case is specified with sequences of actions, the translation of these actions can be defined too. In the same way, the values of the low-level variables that come from the sensors of the robot will be translated into high-level predicates. These translations are done by means of if/then statements; depending on the value of a low-level variable, predicates can be added or deleted from the state of the problem. The interface takes such conditions input by the user and converts them into the high-to-low and low-to-high translation files as described in (González et al. 2018). This way, the output of the interface can be directly integrated into the architecture of the robot, as detailed in the next section. Different catalogues describing what the robot can do are also provided by its designers as input. These catalogues are in comma separated values (csv) format and, while some of them are common to all robots, others are particular for certain platforms. The currently developed catalogues are divided in:

- LowActions: Includes all the low-level instructions that the robot can execute. Each line includes the name of the action and the required parameters separated by commas. Users can choose among these actions to detail the decomposition of the high-level operators.
- Variables: Includes the low level variables that the robot perceives from its sensors. Users have to specify what would happen depending on the values of these variables as a series of conditions. It could be to add a PDDL predicate to the current state, to delete it, or to change the value of the numerical predicates of the state. Those variables are identified by an initial $ symbol.
- Speech: Optional catalogue to specify the possible utterances that the robot can say when speaking. For each line, the id of the speech, its type, and the actual text are specified. This catalogue is only necessary if the robot has the ability to speak and a low-level action to do so, where one of the parameters is the type of speech to say.
- Animations: Optional catalogue to define the animations implemented in the robot. They will be used as parameters of a low-level action to execute the animations.

These catalogues will be associated with a type of robot, so that when users design a use case with the interface they will be able to choose among several available robots. This project currently has catalogues for low actions, variables, and speech.

7 | Evaluation

DeVPlan has undergone different evaluations to demonstrate each of the contributions explained so far, testing their functionality under typical usage and execution. The evaluations are divided into two main parts:

- Evaluation of the graphical user interface: This involved testing the usability of the graphical user interface with a group of inexperienced users. The objective was to assess the differences between hand-coding planning tasks and using the proposed method.
- Evaluation of the integration in a real robotic platform: This phase included injecting two use cases and the translation files required to set up the deliberative architecture into the CLARC robot as generated by the framework.

Subsections below detail the evaluations carried out and their main conclusions.

7.1 | Evaluation of DeVPlan

The usability of the presented framework has been tested with real users. The experimental results reported in this section pursue two main objectives: (1) to evaluate the usability of DeVPlan in comparison to manually handwriting PDDL descriptions and (2) to assess whether the workflow representation is intuitive to depict Social Robotics use cases, making its development feasible even for non-expert PDDL programmers. We recruited a total of 54 fourth-year computer engineering students, working in pairs. While they possess advanced knowledge of programming, they had no prior experience with Automated Planning (AP) or PDDL. As part of a 6 ECTS[1] Knowledge Engineering course, they received 10 h of training in AP and were instructed to use DeVPlan to create different domains and problems. Using DeVPlan, domain experts from retirement homes actively collaborate with knowledge engineers during the modelling process, but it is the latter who take the lead in creating the use case models, particularly when starting from scratch, while retirement home specialists contribute domain expertise to ensure the models align with real-world scenarios. This collaborative approach allows specialists to make changes to the model, such as modifying problem objectives or adding preconditions to actions, without needing to understand the syntax of PDDL. This design lowers the barrier to participation and empowers domain experts to adapt models to evolving requirements independently. For the evaluation presented in this paper, the models were created from scratch, requiring expertise in knowledge engineering. For this process, we used students from a knowledge engineering course to simulate the role of knowledge engineers.

7.1.1 | Protocol

To adapt the experiments to time and resource constraints, we provided the participants with two simple problems to model, referred to as problem A and problem B. We divided participants into two groups. Group 1 (15 pairs) solved problem A directly in PDDL and problem B in the graphical interface, while Group 2 (12 pairs) did the reverse. This approach ensured that

all participants experienced both conditions, enabling a more meaningful analysis of the responses to the questionnaires. We presented them with the following problem definitions:

Problem A. There is a robotic arm in a factory with the task of picking up heavy boxes from a table and placing them onto a conveyor belt. Currently, the arm is teleoperated, but the goal is to automate its behaviour. Whenever the arm is free, it should pick up one of the boxes from the table, holding the box securely. After that, it will rotate to position the box on the conveyor belt. This process continues until all the boxes on the table are placed on the belt.

Problem B. A restaurant wants to introduce a new waiter robot to help in customer service. Initially, the robot is at the bar and is responsible for taking note of each customer's order, which can be either coffee or tea. Coffee orders are prepared in the coffee maker, located at the buffet area, while tea orders are made using the kettle in the kitchen. Once the robot is at the respective preparation area, it should make the coffee or tea and return to the bar to serve it to the corresponding customer.

The experimental procedure is divided in five stages, shown in Figure 11. Before starting the process, participants received an introductory session on AP theoretical concepts. In the next stage, Group 1 was tasked with encoding problem A in PDDL, while Group 2 did the same but in DEVPLAN. Following this, Group 1 switched to DEVPLAN to implement Problem B, while Group 2 coded Problem B in PDDL. Between these two phases, participants were asked to complete questionnaires about their experiences in developing the use cases (with one response submitted per pair within the group).

All tests followed the System Usability Scale (Brooke et al. 1996), where the questions were rated by the participants on a Likert scale ranging from 1 (strongly disagree) to 5 (strongly agree). Table 1 shows the questionnaire for both groups after each implementation, with the aim of identifying differences between developing Social Robotics use cases in DEVPLAN or hand-coding them in PDDL. The questionnaire shown in Table 2 was used to assess the usability of the proposed framework. Our evaluation focuses on a qualitative analysis rather than presenting quantitative data such as time savings or costs. This approach was chosen because our primary aim is to assess the usability and conceptual effectiveness of DEVPLAN in enabling domain experts and knowledge engineers to design robotic use cases. Quantitative metrics, while valuable, would require additional studies under controlled experimental conditions, which were beyond the scope of this initial evaluation.

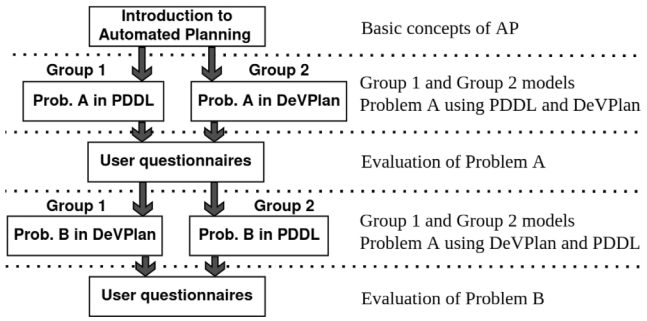


FIGURE 11 | Experimental protocol.

TABLE 1 | Questionnaire to measure the gap between both groups.

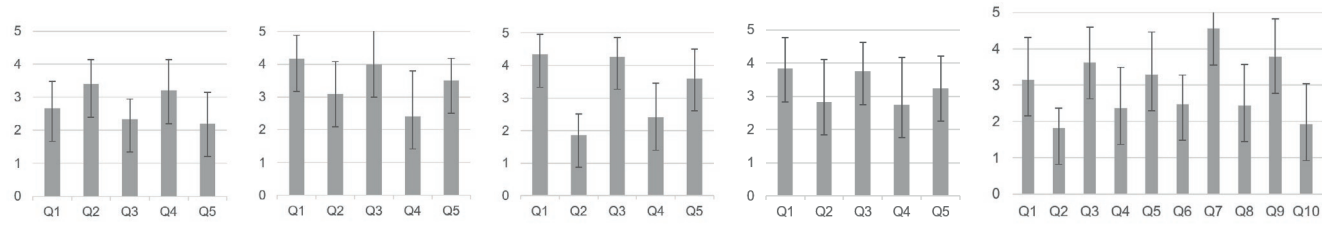
ID	Question
Q1	I think the Social Robotics use case was easy to implement
Q2	It took a long time for me to implement the Social Robotics use case
Q3	I felt very confident implementing the Social Robotics use case
Q4	It took several tries for me to develop the Social Robotics use case
Q5	I could formalise the Social Robotics use case without PDDL knowledge

TABLE 2 | Questionnaire to measure the framework's usability.

ID	Question
Q1	I would like to use this interface frequently
Q2	I found the interface unnecessarily complex
Q3	I thought the interface was easy to use
Q4	I would need the support of a technical person to use this interface
Q5	I found the various functions in this interface were well integrated
Q6	I thought there was too much inconsistency in this interface
Q7	I imagine that most people would learn to use this interface very quickly
Q8	I found the interface very cumbersome to use
Q9	I felt very confident using the interface
Q10	I needed to learn a lot of things before I could use this interface

7.1.2 | Results

Figure 12 shows the results of the questionnaires given to the participants after the implementation of the use cases. Problem A was the first approach to use case modelling, reporting significant differences between Group 1, who hand-coded it, and Group 2, who used DEVPLAN, especially in terms of effort, confidence, and knowledge. It is also worth noting that Group 1, who started implementing problem A in PDDL, had a better experience implementing problem B in the interface the next day than the people who directly started using the interface, who took more time in the implementation. Since the framework is based on AP, it would be interesting to consider such results to pre-train final users in the basic notions of the language.



(a) P. A Group 1 (b) P. A Group 2 (c) P. B Group 1 (d) P. B Group 2 (e) Evaluation of DEVPLAN

FIGURE 12 | Results of the usability questionnaires given to the recruiters. Figure 12a–d shows the scores for each problem A and B according to the implementation carried out in Groups 1 and 2, answering the questions in Table 1. Figure 12e depicts the results of the assessment of DEVPLAN, in which both Group 1 and Group 2 participated together to answer questions shown in Table 2.

Nevertheless, during the first use of the given system, most participants found that DEVPLAN was simple and effortless to use, despite the unfamiliarity. It means that people who are used to programming have found the tool useful and comfortable, even in a similar way as if they were actually programming, but without the disadvantage of having to learn a new programming language.

We believe these are promising results for future evaluations with final users. Since fewer of them are expected to be skilled at programming, we can assume larger differences between the two approaches, especially expecting that most of them will not solve the problem by hand coding it.

7.2 | Field Trials Using CLARC Robot

The last objective of this work is to verify if the generated models are indeed functional once injected into the control architecture. This section summarises the deployment of two real use cases in a retirement home, using DEVPLAN to design the models and integrate them into a cognitive architecture.

We carried out such tests in the context of an EU DIH-Hero project and a national research project related to evaluate the acceptability and utility of a socially assistive robot working in a retirement home. The use cases performed by the robot were designed following a co-creative process involving all end users, which highlighted the importance of optimising the time of healthcare professionals on their daily tasks in a retirement home. Activities such as informing patients about the meals of the day or providing residents with the opportunity to talk to their relatives require staff members to incorporate these tasks into their daily schedules, which are often rigid and difficult to reconcile. The residence has already deployed a CLARC robot (Figure 9), previously used in Comprehensive Geriatric Assessment procedures (Voilmy et al. 2017). In such scenarios, two use cases were initially proposed to be teleoperated or implemented through state machines. The first option was discarded because it still required a person to control the robot manually. State machines seemed to be a better option, but they are tough to implement and update, being also difficult to understand by general users, requiring expertise in programming. By contrast, our no-code proposal based on AP provides an easier and more flexible implementation.

The system was fully tested on the CLARC robot according to the integration procedure explained in Section 6, exposing the generated plans to execution in a real environment. It took less than a week to perform the high-level design, implementation, and testing of the two use cases at the Vitalia Teatinos Residence in Málaga, Spain. The generated files from DEVPLAN were injected into the control architecture, where high-level planning is performed via the Metric-FF (Hoffmann 2003) PDDL2.1 compliant planner. During the use cases, the robot interacted with 5 people with no experience in activities with robots, in addition to residents located in the corridors and halls. In all executions, the robot was totally autonomous, including navigation. Although initially some executions failed and required a manual restart of the platform, by the third day all interactions were correctly executed as described below.

FIGURE 13 | CLARC robot during the announcer use case testing at the retirement home.

Announcer. It was the simplest use case and the first one implemented. The residence has a pre-established monthly lunch and dinner menu with different options depending on the required diet. The proposal is to have a robot in charge of announcing the menu when lunchtime or dinnertime approaches. It is waiting in the charging base until it is time to announce the menu. Then, the robot goes to the main room and plays a sound to notify its arrival (Figure 13). The menu is stated by the robot twice. After that, it goes back to the charging base. Its design contains two main points where the robot can be: the charging base and the hall where the menu has to be announced. Given the generated domain and problem, the planner returns as solution the grounded sequence of actions:

```
0: (MOVE CHARGING_BASE HALL_ANNOUNCE)
1: (PLAY_SOUND HALL_ANNOUNCE)
2: (SAY_MENU HALL_ANNOUNCE)
3: (MOVE HALL_ANNOUNCE CHARGING_BASE)
```

They represent the high level actions taken by the robot, which are sent in commands to the robotic platform. Such decomposition is detailed in Figure 14. It represents a good example of how the same high level action may implement different behaviours in its corresponding low level, depending on the current situation. In this case, the movement has a particularity if the point (represented as p2) to which it is directed is the charging base, where the robot will also introduce a speech called "rest", which signals the end of the use case and indicates that the robot is going to charge.

Videocall. In this use case, the robot first announces the incoming video call in a set of locations, then goes to a specific location and waits for the resident to approach, starting the video call when a person is detected in front of the robot (Figure 15). For this demonstration, we firstly establish various halls where the robot can announce the upcoming video call. Residents of the retirement home spend the day at different places according to their level of dependency. This information is an input to

```
High: move(p1, p2)
Lows: print("MOVE TO " + $p2)
      move($p2)
High: move(p1, p2), $p2 is charging_base
Lows: print("MOVE TO " + $p2)
      say("rest")
      move($p2)
High: say_menu(point)
Lows: print("SAY_MENU " + $point)
      say("menu")
High: play_sound(point)
Lows: print("PLAY_SOUND")
      playSound()
```

FIGURE 14 | High to low level decomposition for the announcer use case.

FIGURE 15 | CLARC robot during a videocall test, where the tablet shows the image of the family member on call.

```
0: (MOVE CHARGING_BASE HALL_ANNOUNCE)
1: (CALL_PATIENT HALL_ANOUNCE PATIENT01)
2: (MOVE HALL_ANOUNCE HALL_CALL)
3: (DETECT_PATIENT PATIENT01 HALL_CALL)
4: (IDENTIFY_PATIENT PATIENT01)
5: (START_VIDEOCALL PATIENT01)
6: (FINISH_VIDEOCALL PATIENT01)
7: (SAY_BYE PATIENT01)
8: (MOVE HALL_CALL CHARGING_BASE)
```

FIGURE 16 | Resulting plan for the videocall use case.

build the initial state, so when the robot receives a petition to perform a video call, it already knows where to go to announce it. The other location previously established is the hall where the call will be held, which is determined by the retirement home. The initial plan generated is shown in Figure 16, where the call is assumed to be executed normally.

But in this use case, it was also tested an example of a change in the expected state of the world: if the call gets cancelled, the robot receives the current new state and replans accordingly, stopping the use case and returning to the charging base. Figure 17 shows the use case as implemented in DEVPLAN.

Although from an AP point of view, the generated plans are simple (low number of actions), for more complex scenarios our approach would also provide shorter deployment times compared to other techniques. In just 1 week, we were able to design two prototypes, integrate the control architecture on the robotic platform, and test both use cases in a real environment. With all components working and MLARAS integrated into the robotic platform, this time can be reduced for future developments, as only the PDDL domain and problem definition need to be changed.

In summary, the use of deterministic planners addresses issues that may arise from execution in stochastic environments. This is achieved through a replanning process that utilises recovery options and checkpoints. The speed of modern deterministic planners enables efficient real-world executions, as demonstrated by successful real-time operations in the retirement home.

8 | Discussion

Our work in the field of Social Robotics domain modelling aims to simplify the development process by defining the tasks that the robot can perform as options. These options can be combined to create complex behaviours that fulfil the objectives of the use case. In comparison to other modelling approaches, such as Finite State Machines, DEVPLAN generates models that are not only easier to create but also easier to comprehend.

A use case similar to the video call was developed for a hospital where, due to COVID restrictions, the robot also needs to be disinfected after each call. This use case was implemented through the state machine shown in Figure 18. This method requires specifying all system transitions, making it difficult to model and lacking the ability to generalise. In fact, the video call system was previously implemented for this purpose. However, creating

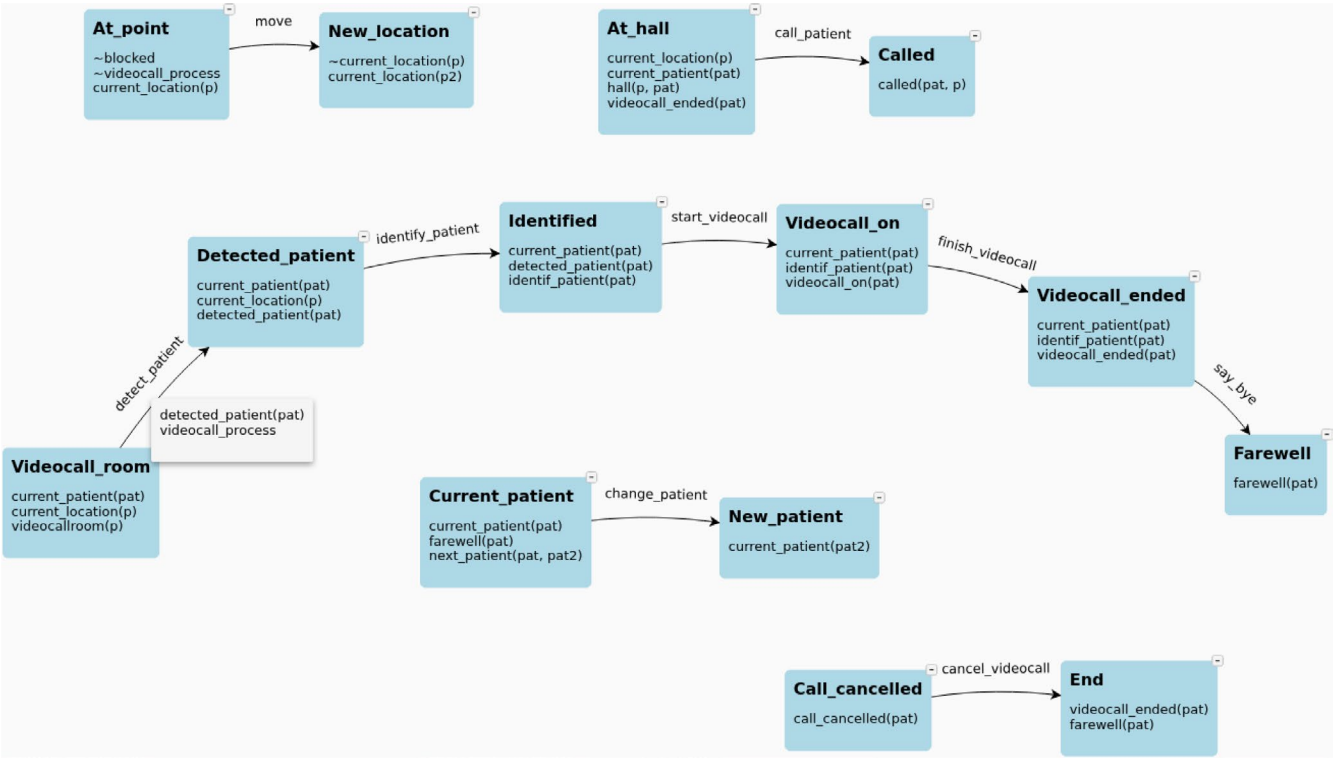


FIGURE 17 | Videocall use case modelled in the graphical interface.

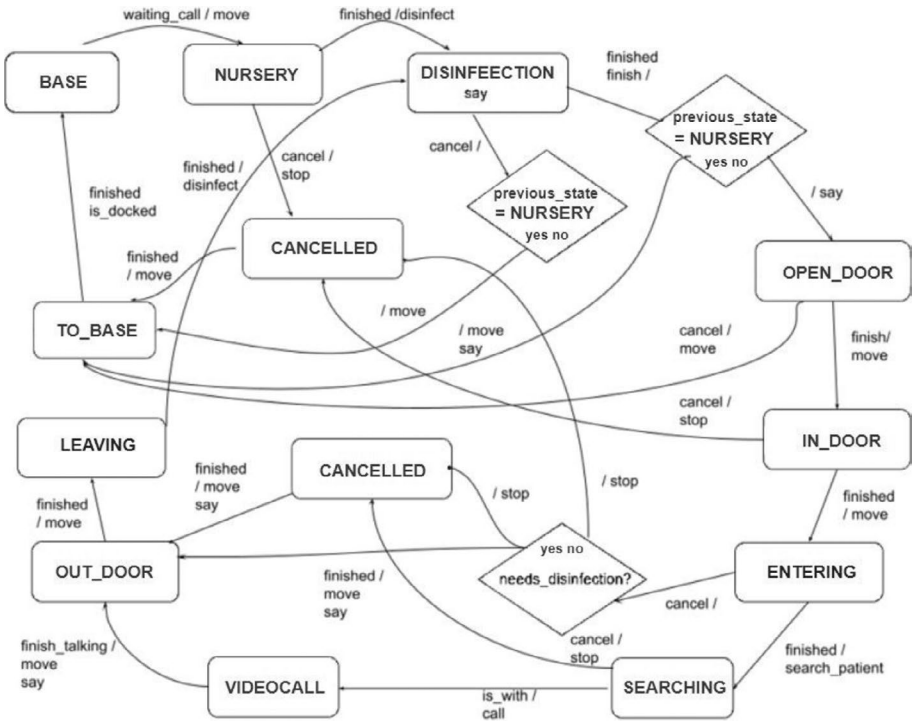


FIGURE 18 | Graphical representation of the Finite State Machine for the hospital videocall use case. All possible transitions must be specified.

a new state machine to adapt the use case to the requirements of the retirement home (e.g., no disinfection, announcement of the upcoming video call, etc.) was challenging due to the complexity of its development. By contrast, in Figure 19 we present a problem modelling approach that closely resembles an intuitive representation of the main tasks of the use case. Here, the options are clearly specified, including navigation, requesting disinfection, managing a cancelled call, and the general video call process. This representation is also comprehensive enough to be valid for multiple calls with different patients and to detect blocking objects at any time (modelled as exogenous events). To illustrate the capabilities of this approach, and in contrast

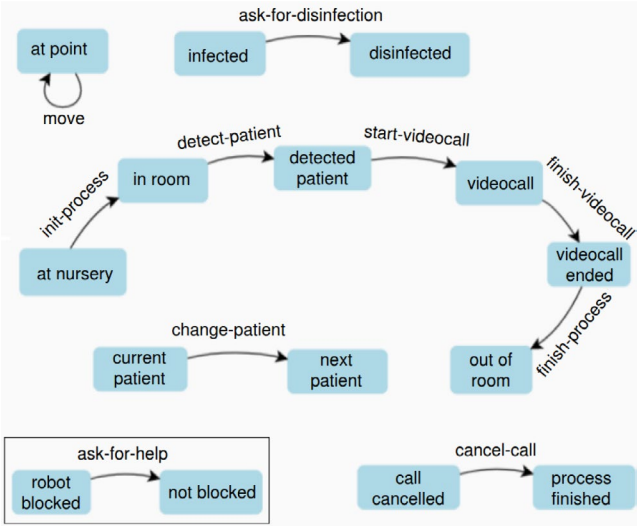


FIGURE 19 | Graphical representation for the hospital videocall using DevPlan. Unexpected events are handled by the control architecture using AP.

to the state machine implementation, the model generated in DEVPLAN was effortlessly translated to the use case needed at the retirement home, as shown in Figure 17.

9 | Related Work

In this work we propose a link between the graphical workflows and the PDDL formalism through the definition of *options*. Such concept is borrowed from (Sutton et al. 1999) and is based on *macro-operators* (Korf 1985): sets of actions that are usually applied sequentially in a plan. In contrast to them, which specify a sequence of actions that has to be executed as a whole, options can be partially executed and resumed from any point. In addition, macro-operators focus on the actions applied, abstracting out the partial states traversed, while in our approach both actions and states are equally important and must be specified by the user. Options have some similarities with timelines (Jónsson et al. 2000) and partial plans (Minton et al. 1994; Weld 1994). A timeline is a temporal description of the different values a state variable takes. In timeline-based planning the use case is modelled in terms of a series of state variables and temporal constrains among their values. For example, variable A can only take value a_i, after variable B has taken value b_j. Initial state and goals are expressed as current and future values of some of the variables, respectively. The planner has then to *fill the gaps* in the timelines to reach the desired values. In a similar way, our planner has to *fill the gaps* in the options interleaving other options. The main difference is that there is no concept of *action* in timeline-based planning, while in our approach actions are a crucial element. Options can also be seen as totally ordered partial plans, as actions in each option must appear in the plan one after the other, and there can be some other actions in between. But partial plans do not make any assumption about the states the plan traverses, while options imply to reach the states included in them. Behaviour Trees are also employed in literature as an alternative to FSM. However, they primarily lead to very reactive behaviours, which is why they are often combined with planning approaches (Neufeld et al. 2018; Colledanchise et al. 2019). These planning approaches fulfil the primary requirement of our work, which is strategic planning. In our scenario, reactive components are necessary when a plan fails and are managed through a replanning process that adapts to the plan to the new situation.

Related to the knowledge engineering process behind the development of AP models for Social Robotics systems, various tools have been introduced to support the implementation of such planning domains. In addition to PDDL editors[2], which require deep knowledge about the specification language, we can find in the literature systems characterised by automatically translating the resulting visual model into its PDDL formalisation (Vaquero et al. 2013; Simpson et al. 2007; Hatzi et al. 2010). Although all of these systems use different graphical representations to specify planning domains, they focus on users with a deep knowledge of software engineering and become unmanageable for large domains. Instead, the design and implementation of robotic platforms are usually covered by specific toolkits (Pot et al. 2009; Touretzky and Tira-Thompson 2011; Kim and Jeon 2007; Jackson 2007). Some of them provide visual programming utilities for novice users, but they are restricted to hardware configurations or basic programming, not being able to build general models. So a major missing feature in current robotic development tools is the possibility to have both visual programming and general model generators, in addition to mechanisms to properly manage the human-robot interaction. These are some of the highlighted features of the interface we propose.

10 | Conclusions and Future Work

Automated Planning has been reported in the literature as a general approach to SAR development. However, it is not extensively used, primarily due to factors such as the bottleneck in model development, which often requires extensive knowledge engineering processes. In this paper, we propose DEVPLAN, a tool that allows non-experts to participate in the design of complex real-world use cases by defining the possible *options* to be executed by the robotic platform. Users can depict the expected behaviour of the robotic platform through simple conducts, which can be interleaved to create more sophisticated and robust behaviours. This representation of *options* offers a simpler and more versatile approach compared to other techniques such as FSM's, particularly in complex use cases or situations where there is no fixed sequence of actions to solve the use case. Additionally, DEVPLAN generates files to set up the control architecture embedded in the robotic platform, eliminating the need for hard-coding components that would otherwise require modification for each unique use case. The evaluation carried out evidences that DEVPLAN results useful to create social robotics use cases, and that the generated models are executable once injected into a real robotic platform.

Currently, the described use cases are launched manually, having to decide which one of them the robot must perform at any time. We are currently working on implementing an additional layer of deliberation to automate the scheduling of use cases throughout the day, while also identifying opportunities for the robot to perform smaller tasks between them. As part

of our future work, we plan to deploy the system in production, where real-world data can offer quantitative insights into its performance and impact. Another avenue for improvement is integrating large language models (LLMs) into the DEVPLAN interface, aligning with the recent trend of combining LLMs with automated planning (Pallagani et al. 2024). Such integration could enhance the system by predicting and suggesting model elements based on user inputs, thereby simplifying the model creation process and significantly improving the user experience.

Author Contributions

Alba Gragera, Carmen Díaz de Mera, Fernando Fernández, and Ángel García Olaya conducted the study and designed the proposed framework. The code was implemented by Alba Gragera and Carmen Díaz de Mera. The robotic platform was deployed and tested by Alba Gragera, Carmen Díaz de Mera, and Juan Pedro Bandera. All authors contributed to writing this manuscript, and all authors have read and approved the final manuscript.

Acknowledgements

This work has been funded by the European Union's Horizon 2020 research and innovation program under grant agreement No 825003 (DIH-HERO SUSTAIN), by the ECHORD++ (FP7-ICT-601116) project, by grants RTI2018-099522-B-C41, TED2021-131739B-C21, PID2021-127647NB-C21, PID2022-137344OB-C32, PDC2022-133597-C42 and PDC2022-133597-C43 from MICIU/AEI/10.13039/501100011033, by the ERDF "A way of making Europe" and Next Generation EU/PRTR, and by the Madrid Government under the Multiannual Agreement with UC3M in the line of Excellence of University Professors (EPUC3M17) in the context of the V PRICIT (Regional Program of Research and Technological Innovation).

Ethics Statement

The study was conducted according to the guidelines of the Declaration of Helsinki and approved by the Andalusian Ethics Committee.

Consent

Informed consent was obtained from all subjects involved in the study.

Conflicts of Interest

The authors declare no conflicts of interest.

Data Availability Statement

All data generated or analysed during this study are included in this published article. The data sources (use case recordings, interviews, etc.) employed to generate these data are not publicly available because they contain sensible personal information. However, some of them are available on reasonable request.

Endnotes

[1] European credits: 1 credit is equivalent to 25 h of student workload.

[2] http://editor.planning.domains/.

References

Bandera, A., J. P. Bandera, P. Bustos, et al. 2016. *Clarc: A Robotic Architecture for Comprehensive Geriatric Assessment*, 1–8. WAF.

Bhatnagar, S., S. Mund, E. Scala, K. McCabe, T. L. McCluskey, and M. Vallati. 2022. "On-the-Fly Knowledge Acquisition for Automated Planning Applications: Challenges and Lessons Learnt." In *Proceedings of the 14th International Conference on Agents and Artificial Intelligence, (ICAART)*, edited by A. P. Rocha, L. Steels, and H. J. van den Herik, vol. 2, 2022, 387–397. SCITEPRESS.

Bolander, T., and M. B. Andersen. 2011. "Epistemic Planning for Single and Multi-Agent Systems." *Journal of Applied Non-Classical Logics* 21, no. 1: 9–34.

Bonet, B., and H. Geffner. 2013. "Causal Belief Decomposition for Planning With Sensing: Completeness Results and Practical Approximation." In *International Joint Conference on Artificial Intelligence*, 2275–2281. IJCAI/AAAI.

Borrajo, D., and M. Veloso. 2012. "Probabilistically Reusing Plans in Deterministic Planning." In *Proceedings of ICAPS'12 Workshop on Heuristics and Search for Domain-Independent Planning*, 17–25.

Breazeal, C., K. Dautenhahn, and T. Kanda. 2016. "Social Robotics." In *Springer Handbook of Robotics*, edited by B. Siciliano and O. Khatib, 1935–1972. Springer Handbooks.

Brooke, J., P. Jordan, B. Thomas, B. Weerdmeester, and I. McClelland. 1996. *Usability Evaluation in Industry*. CRC Press.

Bustos, P., L. J. Manso, A. J. Bandera, J. P. Bandera, I. García-Varea, and J. Martínez-Gómez. 2019. "The Cortex Cognitive Robotics Architecture: Use Cases." *Cognitive Systems Research* 55: 107–123.

Bustos, P., L. J. Manso, J. P. B. Rubio, et al. 2015. "A Unified Internal Representation of the Outer World for Social Robotics." In *Robot 2015: Second Iberian Robotics Conference - Advances in Robotics, Lisbon, Portugal, 19–21 November 2015, Volume 2. Advances in Intelligent Systems and Computing*, vol. 418, 733–744. Springer.

Cashmore, M., M. Fox, D. Long, et al. 2015. *Rosplan: Planning in the Robot Operating System*, 333–341. ICAPS.

Celorrio, S. J., F. Fernández, and D. Borrajo. 2008. "The {PELA} Architecture: Integrating Planning and Learning to Improve Execution." In *Proceedings of the Twenty-Third {AAAI} Conference on Artificial Intelligence*, edited by D. Fox and C. P. Gomes, 1294–1299. AAAI Press. http://www.aaai.org/Library/AAAI/2008/aaai08-205.php.

Chen, K., F. Yang, and X. Chen. 2016. "Planning with Task-Oriented Knowledge Acquisition for a Service Robot." In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, edited by S. Kambhampati, 812–818. IJCAI/AAAI Press.

Colledanchise, M., D. Almeida, and P. Ögren. 2019. "Towards Blended Reactive Planning and Acting Using Behavior Trees." In *International Conference on Robotics and Automation, ICRA 2019*, 8839–8845. IEEE.

Fox, M., A. Gerevini, D. Long, and I. Serina. 2006. "Plan Stability: Replanning Versus Plan Repair." In *International Conference on Automated Planning and Scheduling*, 212–221. AAAI.

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

García-Olaya, A., R. Fuentetaja, J. García-Polo, J. C. González, and F. Fernández. 2019. "Challenges on the Application of Automated Planning for Comprehensive Geriatric Assessment Using an Autonomous Social Robot." In *Advances in Intelligent Systems and Computing*, 179–194. Springer International Publishing.

Geffner, H., and B. Bonet. 2013. "A Concise Introduction to Models and Methods for Automated Planning." In *Synthesis Lectures on Artificial Intelligence and Machine Learning*. Morgan & Claypool Publishers.

Ghallab, M., D. Nau, and P. Traverso. 2004. *Automated Planning: Theory & Practice*. Elsevier.

González, J. C., J. Garcia, R. Fuentetaja, A. Olaya, and F. Fernández. 2018. "From High to Low Level and Vice-Versa: A New Language

for the Translation Between Abstraction Levels in Robot Control Architectures." In *3rd Workshop on Semantic Policy and Action Representations for Autonomous Robots (SPAR)*.

González, J. C., Á. García-Olaya, and F. Fernández. 2020. "Multi-Layered Multi-Robot Control Architecture for the Robocup Logistics League." In *2020 IEEE International Conference on Autonomous Robot Systems and Competitions, ICARSC 2020, Ponta Delgada, Portugal, April 15-17*, 120–125. IEEE.

González, J. C., J. C. Pulido, and F. Fernández. 2017. "A Three-Layer Planning Architecture for the Autonomous Control of Rehabilitation Therapies Based on Social Robots." *Cognitive Systems Research* 43: 232–249.

Hatzi, O., D. Vrakas, N. Bassiliades, D. Anagnostopoulos, and I. P. Vlahavas. 2010. "A Visual Programming System for Automated Problem Solving." *Expert Systems With Applications* 6: 4611–4625.

Hoffmann, J. 2003. "The Metric-FF Planning System: Translating "Ignoring Delete Lists" to Numeric State Variables." *Journal of Artificial Intelligence Research* 20: 291–341.

Ingrand, F., and M. Ghallab. 2017. "Deliberation for Autonomous Robots: A Survey." *Artificial Intelligence* 247: 10–44.

Jackson, J. 2007. "Microsoft Robotics Studio: A Technical Introduction." *IEEE Robotics and Automation Magazine* 14, no. 4: 82–87.

Jónsson, A. K., P. H. Morris, N. Muscettola, K. Rajan, and B. D. Smith. 2000. "Planning in Interplanetary Space: Theory and Practice." In *Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems, Breckenridge, CO, USA, April 14-17, 2000*, edited by S. A. Chien, S. Kambhampati, and C. A. Knoblock, 177–186. AAAI.

Kambhampati, S. 2007. "Model-lite Planning for the Web Age Masses: The Challenges of Planning with Incomplete and Evolving Domain Models." In *Proceedings of the Twenty-Second {AAAI} Conference on Artificial Intelligence, July 22-26, 2007, Vancouver, British Columbia, Canada*, 1601–1605. AAAI Press.

Kim, S. H., and J. W. Jeon. 2007. *Programming Lego Mindstorms Nxt With Visual Programming*, 2468–2472. ICCAS.

Korf, R. E. 1985. "Macro-Operators: A Weak Method for Learning." *Artificial Intelligence* 26, no. 1: 35–77.

Kramer, J. F., and M. Scheutz. 2007. "Development Environments for Autonomous Mobile Robots: A Survey." *Autonomous Robots* 2: 101–132.

Manso, L., P. Bachiller, P. Bustos, P. Núñez, R. Cintas, and L. Calderita. 2010. *Robocomp: A Tool-Based Robotics Framework 6472*, 251–262. Springer.

Martínez, J., A., Romero-Garcés, C., Suárez. et al. 2018. "Towards a Robust Robotic Assistant for Comprehensive Geriatric Assessment Procedures: Updating the {CLARC} system." *27th {IEEE} International Symposium on Robot and Human Interactive Communication, {RO-MAN} 2018, Nanjing, China, August 27-31, 2018*, 820–825. IEEE.

McCluskey, T. L., T. S. Vaquero, and M. Vallati. 2017. "Engineering Knowledge for Automated Planning: Towards a Notion of Quality." In *Proceedings of the Knowledge Capture Conference, K-CAP 2017*, edited by Ó. Corcho, K. Janowicz, G. Rizzo, I. Tiddi, and D. Garijo, 14–1148. ACM.

McDermott, D., M. Ghallab, A. Howe, et al. 1998. *PDDL-The Planning Domain Definition Language*.

McGann, C., F., Py, K., Rajan, et al. 2008. *Adaptive Control for Autonomous Underwater Vehicles*, 1319–1324.

Minton, S., J. Bresina, and M. Drummond. 1994. "Total-Order and Partial-Order Planning: A Comparative Analysis." *Journal of Artificial Intelligence Research* 2: 227–262.

Mohseni-Kabir, A., M. Veloso, and M. Likhachev. 2020. "Efficient Robot Planning for Achieving Multiple Independent Partially Observable Tasks That Evolve over Time." In *Proceedings of the Thirtieth International Conference on Automated Planning and Scheduling, Nancy, France, October 26-30, 2020*, edited by J. C. Beck, O. Buffet, J. Hoffmann, E. Karpas, and S. Sohrabi, 202–211. AAAI Press.

Muise, C. J., V. Belle, and S. A. McIlraith. 2014. "Computing Contingent Plans via Fully Observable Non-deterministic Planning." *Proceedings of the AAAI Conference on Artificial Intelligence* 28, no. 1: 2322–2329. https://doi.org/10.1609/aaai.v28i1.9049.

Neufeld, X., S. Mostaghim, and S. Brand. 2018. "A Hybrid Approach to Planning and Execution in Dynamic Environments Through Hierarchical Task Networks and Behavior Trees." In *Proceedings of the Fourteenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE 2018*, edited by J. P. Rowe and G. Smith, 201–207. AAAI Press.

Pallagani, V., B. C. Muppasani, K. Roy, et al. 2024. "On the Prospects of Incorporating Large Language Models (Llms) in Automated Planning and Scheduling (APS)." In *Proceedings of the Thirty-Fourth International Conference on Automated Planning and Scheduling, ICAPS 2024*, edited by S. Bernardini and C. Muise, 432–444. AAAI Press.

Pot, E., J. Monceaux, R. Gelin, and B. Maisonnier. 2009. "Choregraphe: A Graphical Tool for Humanoid Robot Programming." In *18th {IEEE} International Symposium on Robot and Human Interactive Communication, {RO-MAN} 2009, Toyama International Conference Center*, 46–51. IEEE.

Puterman, M. L. 1994. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley Series in Probability and Statistics. Wiley.

Rajan, K., and F. Py. 2012. "T-REX: Partitioned Inference for AUV Mission Control." In *Further advances in unmanned marine*, 171–199. Institution of Engineering and Technology IET.

Rintanen, J. 2004. "Complexity of Planning With Partial Observability." In *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS 2004)*, edited by S. Zilberstein, J. Koehler, and S. Koenig, 345–354. AAAI.

Sanner, S. 2011. Relational dynamic influence diagram language (RDDL): Language description. In: Proceedings of the Seventh International Planning Competion.

Simpson, R. M., D. E. Kitchin, and T. L. McCluskey. 2007. "Planning Domain Definition Using GIPO." *Knowledge Engineering Review* 2: 117–134.

Sutton, R. S., D. Precup, and S. P. Singh. 1999. "Between mdps and Semi-mdps: A Framework for Temporal Abstraction in Reinforcement Learning." *Artificial Intelligence* 112, no. 1–2: 181–211.

Tapus, A., M. J. Mataric, and B. Scassellati. 2007. "Socially Assistive Robotics [Grand Challenges of Robotics]." *IEEE Robotics and Automation Magazine* 1: 35–42.

Touretzky, D. S., and E. J. Tira-Thompson. 2011. *The Tekkotsu Robotics Development Environment*, 6084–6089. ICRA.

Tran, T. T., T. S. Vaquero, G. Nejat, and J. C. Beck. 2017. "Robots in Retirement Homes: Applying Off-The-Shelf Planning and Scheduling to a Team of Assistive Robots." *Journal of Artificial Intelligence Research* 58: 523–590.

Vaquero, T. S., J. R. Silva, F. Tonidandel, and J. C. Beck. 2013. "itSIMPLE: Towards an Integrated Design System for Real Planning Applications." *Knowledge Engineering Review* 28, no. 2: 215–230.

Voilmy, D., C. Suarez, A. Romero-Garcés, et al. 2017. "CLARC: A Cognitive Robot for Helping Geriatric Doctors in Real Scenarios." In *ROBOT 2017: Third Iberian Robotics Conference - Volume 1, Seville, Spain, November 22–24, 2017. Advances in Intelligent Systems and Computing*, vol. 693, 403–414. Springer.

Weld, D. S. 1994. "An Introduction to Least Commitment Planning." *AI Magazine* 15, no. 4: 27–61.

Yoon, S. W., A. Fern, and R. Givan. 2007. "Ff-Replan: A Baseline for Probabilistic Planning." In *Proceedings of the Seventeenth International*

Conference on Automated Planning and Scheduling, ICAPS 2007, Providence, edited by M. S. Boddy, M. Fox, and S. Thiébaux, 352–359. AAAI.

Younes, H. L., and M. L. Littman. 2004. "Ppddl1. 0: An Extension to pddl for Expressing Planning Domains With Probabilistic Effects." Technical Report, CMU-CS-04-162 2, 99.