Executing Robot Task Models in Dynamic Environments

Kai Adam, Arvid Butting, Oliver Kautz, Bernhard Rumpe, Andreas Wortmann
Software Engineering, RWTH Aachen University
www.se-rwth.de

***Abstract*—Deploying successful robotics applications requires tremendous effort due to the need for contributions of experts from various domains. We present the iserveU family of executable DSLs that separate the concerns of domain experts and robotics experts and leverage model-transformation at system run-time to enable the robotic platform to flexibly fulfill tasks in a changing real-world environment. Current research in DSLs for robotics applications focuses on abstraction in the solution domain, whereas our DSLs support the domain expert in declaratively describing properties of the domain and loosely coupled tasks. To enable flexible task execution based on the domain expert's declarative models, these are translated into components of a reference architecture prior to deployment and into planning domain definition language (PDDL) problems at system run-time. Resulting problems are translated into executable plans using the Metric-FF solver and re-translated into iserveU models that ultimately are executed against a loosely coupled robotics middleware. Leveraging model transformation at run-time enables the flexibility necessary for robotics applications deployed to dynamic environments where design-time assumptions and run-time reality diverge easily.**

***Index Terms*—Model-Driven Development; Domain-Specific Languages; Planning; Software Architectures; Service Robotics**

I. Motivation

Service robotics is one of the most challenging domains: successful deployment of even simple applications requires participation of experts from various domains including navigation, perception, software engineering, and the application domain. Dynamic environments are a major challenge to service robotics success: robots deployed to such environment must either feature deterministic solutions for many, if not all, possible challenges or yield flexible solution strategies to solve service tasks in the face of unforeseen challenges. To achieve this, robotic experts usually employ planning techniques borrowed from knowledge representation. These techniques, such as Golog [1], are tailored towards knowledge representation experts and purely virtual planning only, which raises two challenges: (1) Enabling domain experts to represent the knowledge required to support run-time problem solving; (2) Realizing the actions of the resulting plan in the real world.

We conceived the iserveU family of domain-specific languages (DSLs) that separate the concerns of domain experts and robotics experts [2]. These DSLs enable domain experts to specify tasks, goals, and domain causalities in DSLs specifically tailored to describing service robotics applications. To this effect, they feature robot entity models that are grounded in robotics middlewares (such as ROS [3]) by respective experts. The iserveU framework transforms entity models into component implementations of its reference architecture [4] and leverages transforming service robotics tasks into planning problems at system run-time to enable the service robot to flexibly fulfill tasks in a changing environment. Based on previous work ([2], [4]) this paper presents

- the execution of iserveU task and goals models by leveraging their run-time transformation into problems of the planning domain definition language (PDDL),
- solving these problems with the Metric-FF planner, and
- transforming the resulting plans into sequences of robot actions that ultimately are executed with a loosely coupled robotics middleware in the real world.

In the following, Sec. II describes preliminaries and Sec. III presents an example of the iserveU DSLs in action. Afterwards, Sec. IV introduces the reference architecture that enables model execution and Sec. V presents the language transformations. Sec. VI highlights related work and Sec. VII discusses our approach. Sec. VIII concludes.

II. Preliminaries

This section recapitulates quintessential concepts of the iserveU DSLs presented in [2] and introduces PDDL as well as the Metric-FF planner.

A. The iserveU DSLs

The iserveU DSLs are a family of four textual DSLs that enable describing platform-independent, reusable robotic tasks. To this effect, we separate the concerns of the domain expert, who knows the domain's types, causalities, and concerns of the robot platform and its environment (denoted as world) [2]. The latter describe the robot's services as tasks that comprise sequences of reusable goals, *i.e.,* conditions over the properties of robot and world that must hold at some point in time.

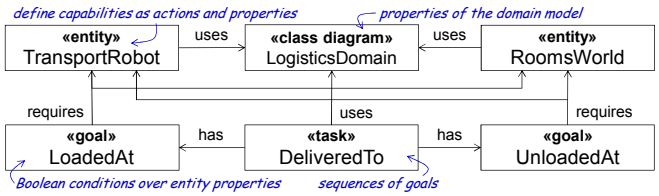

Figure 1. Models of the four iserveU DSLs: CDs, entities, goals, and tasks.

Fig. 1 illustrates the relations of the four iserveU DSLs by example: UML/P [5] class diagram (CD) models describe the

platform-independent concepts of the logistics domain. Those classes are accessible in models representing entities, tasks, and goals of a robotics application and can be manipulated by in-class specified methods. Entity models describe properties and actions of actors (robot, world) that can manipulate the environment and operate in the context of a shared domain. Entity properties yield parameters and return values of types defined in the domain model (plus Java basics). Actions declared in the entity model specify the actor's capabilities in terms of preconditions and postconditions over the entity properties or domain model concepts. They resemble actions of STRIPS [6], which enable planners to reason over action executability and their effects. Goal models describe situations as parametrizable Boolean conditions over domain types and entity properties. Task models are sequences of goals that must hold in the predefined order. They can be parametrized and pass arguments to their goals.

The DSLs and their MontiArcAutomaton [7] reference architecture, have been deployed to and evaluated in the Katharinenhospital in Stuttgart, Germany [2], [4].

B. The Planning Domain Definition Language and Metric-FF

PDDL [8] is a widely used artificial intelligence planning language. It distinguishes domain and problem models. A domain model expresses the properties of a domain, *i.e.,* all relevant objects, predicates, and actions. An action consists of parameters, a precondition over the parameters, and an effect. The precondition may reference the action's parameters and describes when it is executable with respect to a current domain state. Its effect describes the domain changes that occur on action execution. A precondition can be interpreted as an arbitrary function-free first order predicate over the action's parameters. An effect is a list of changes the action imposes on its execution. Effects may contain universal quantification and conditional expressions. As those are simply lists of actions, a universal quantification can be interpreted as a for-each loop. A conditional expression is a simple if-then-else expression. A PDDL problem describes an initial domain state in terms of initially satisfied predicates, *i.e.,* existing objects and the properties they satisfy, and a goal describing the desired world state that is to be achieved. Goals are arbitrary function-free first order predicates also.

Solving PDDL Problems requires an efficient planner. For our implementation, we choose Metric-FF [9], a domain independent and competitive planning system for PDDL. From a black-box perspective, Metric-FF takes a PDDL domain and a PDDL problem as input and outputs a plan, *i.e.,* a list of actions. Starting from the domain state described by the PDDL problem, executing the actions in order leads to satisfaction of the goal described by the PDDL problem.

III. Example

Consider describing robot delivery tasks. Decoupling these from the specific environments they can be executed in and from the platforms they can be executed with, requires proper abstractions. With the iserveU languages, the domain model is abstracted to a CD holding information of the domain's concepts and their relations. To deliver items between rooms, this can be manifested as the CD depicted in Fig. 2.

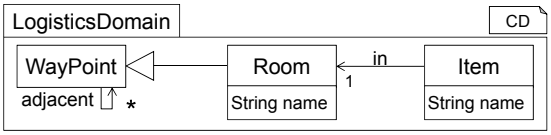

Figure 2. Domain model characterizing items in connected rooms.

This domain model describes the world (*i.e.,* the environment), consisting of connected waypoints out of which some are rooms that store items. It does not disclose who knows about this or can provide instances to reason about. For this, the iserveU languages distinguish actions and properties of robot entities and those defined by the world entities. Worlds provide information about everything required by the robot to fulfill tasks. Consequently, in our example, the world knows where items are and which rooms are adjacent. We model this as the entity model depicted in Fig. 3. The entity model is

```
01 domain LogisticsDomain;                                          Entity
02 world RoomsWorld {
03  property Boolean itemLoc(Item item, Room room);
04  property Boolean adjacent(Waypoint w1, Waypoint w2);
05  /* Additional properties */
06 }
```

Figure 3. World entity providing information about rooms and items.

valid in the context of the aforementioned domain model (l. 1) and begins with the keyword `world` (to distinguish it from robot entity models) followed by its unique name (l. 2) and a body of properties (ll. 3-5). The properties describe relations over the instances characterized in the domain model: here they describe whether an item is in a specific room (l. 3) and whether two waypoints are adjacent (l. 4). Modeling these relations independent of their technical realization (which is bound later during design time [4]) enables reusing these concepts in different environments and with different robots with little effort. Similarly, the robot entity depicted in Fig. 4

```
01 domain LogisticsDomain;                                          Entity
02 world RoomsWorld rw;
03 robot TransportRobot {
04  property Waypoint robotLoc();
05  property Boolean hasLoaded(Item item);
06  action move(Waypoint from, Waypoint to) {
07   pre:  robotLoc() == from && rw.adjacent(from, to);
08   post: robotLoc() == to;
09  }
10  action pickUp(Item item, Room room) {
11   pre:  robotLoc() == room && rw.itemLoc(item, room);
12   post: hasLoaded(item)    && !rw.itemLoc(item, room);
13  }
14  /* Additional properties and actions */
15 }
```

Figure 4. Robot entity model specifying its properties and capabilities.

is valid in the context of a domain and world (ll. 1-2) only. It characterizes properties (ll. 4-5) and actions (ll. 6-14) the robot entity is capable of, independent of their technical realization.

Robots realizing this entity must be able to identify, which waypoint they are at (l. 4) and which item they have loaded (l. 5). How, for instance, localization is realized is irrelevant. Actions feature unique (in context of the containing entity) names and parameters over which they specify preconditions and postconditions. Preconditions describe when executing the action is possible, postconditions describe the effect of executing the action. For instance, picking up an item in a specific room (ll. 10-13) requires that the robot and the item are in that room (l. 11). Here, `itemLoc` references the properties specified in the `RoomsWorld` world entity. On successful action execution, the item is not in that room anymore, but loaded onto the robot (l. 12). These causalities are independent of the specific domain (whether the item is moved around in a factory or hospital is irrelevant) and independent of the employed platforms.

Tasks in the iserveU context are sequences of goals the robot must fulfill given what is specified in the world and entity models. These goals are Boolean conditions over properties specified in world and robot entities. The goal `LoadedAt`, as depicted in Fig. 5, for instance, is considered fulfilled if the related `TransportRobot` instance (l. 1) confirms being in the specified room and holding the specified item (l. 3). Instead of imperatively describing a sequence of actions

```
01 robot TransportRobot rob;                                    Goal
02 goal LoadedAt(Room room, Item item) {
03   (rob.robotLoc() == room) && (rob.hasLoaded(item))
04 }
```

Figure 5. Goal requiring the robot being in a room while holding an item.

that should lead to fulfilling goals at design time, specifying the participating entities and their capabilities declaratively enables finding solutions using a planner at system run time where potentially unforeseen challenges can arise. This yields greater flexibility and more robust robotics applications.

IV. Reference Architecture

The approach relies on a centralized software architecture, which is deployed to a system that is capable of communicating with the robot and the user interface. The software architecture is modeled as MontiArcAutomaton C&C architecture as depicted in Fig. 6. The component `Controller` organizes the orchestration of the overall process of task execution, as described in [4]. If a task entering the architecture is executed, the `Controller` starts by dividing it into an ordered list of goals that the robot has to fulfill one after another. It selects the next goal and sends it to the `Planner` component, which either calculates a plan to achieve the goal or indicates that no plan exists. The plan may depend on the current status of each element of the world and the current status of the robot. Therefore, the `Planner` component can query the `StateProvider` component to obtain current states of robot and world. A valid plan to achieve a goal comprises an ordered list of actions. The `Controller` selects the next action to be executed and sends it to the `ActionExecutor` component.

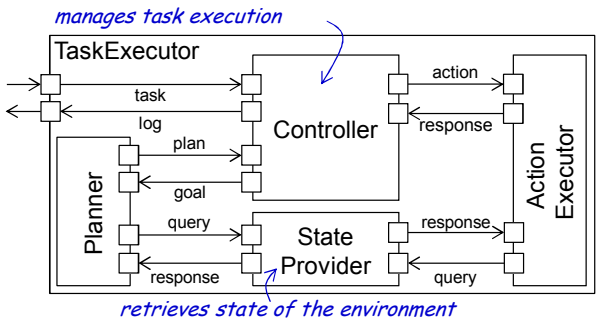

Figure 6. Overview of the iserveU reference architecture.

The execution of actions is influenced by the status of robot and/or world, which is queried from the `StateProvider` component. If the execution of an action is finished, it can be either successful or not. On success, the controller executes the next action, otherwise, it triggers the `Planner` component to calculate an alternative plan. The implementation of the `Controller` component is independent of the concrete models and is therefore part of the run-time system. The `StateProvider` component requires all available properties of robot and world and is thus partially generated from the properties defined within entity models. The implementation of the `ActionExecutor` is generated from actions of entity models. It delegates the actual execution of actions to an employed middleware API. Therefore, a Java interface for each robot and world is generated, and the handwritten implementation of these interfaces calls the employed middleware API. The implementation of the `Planner` component itself is independent of the concrete types of goals it processes and actions it produces. Instead, it is part of the run-time system and invokes a PDDL tool with arguments generated as described in Sec. V.

V. To PDDL and Back Again

Developers use the iserveU DSLs to describe domain knowledge with UML/P CDs, tasks consisting of sequences of goals, and entities consisting of properties and actions. Solving tasks requires deriving actions to satisfy their goals step-by-step. The latter is a classic planning problem, hence we translate entities with actions and properties as well as domain knowledge to PDDL, use Metric-FF [9] to solve each goal, and transform the results back to plans at runtime.

The reference architecture's `Planner` component implements the transformations to calculate plans for goals using the template-based code generation facilities of MontiCore. Fig. 7 overviews the main modules of the planning infrastructure and their dependencies. At design time, application developers model entities with properties and actions, domain model CDs, and goals representing desired situations (Sec. II-A). At compile time, the planning infrastructure applies several preparing model transformations to the entity models to, *e.g.*, ensure uniqueness of names for the predicates of the PDDL artifacts generated later. Based on the transformed entity

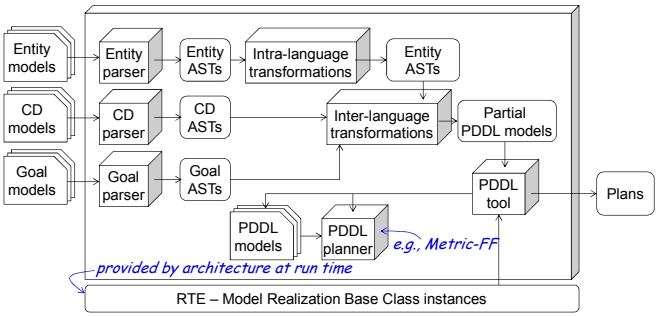

Figure 7. Overview of the transformation infrastructure.

```
01 (define (domain myDomain)                                    PDDL
02  (:types
03   String   - object   ; build-in type
04   Boolean  - object   ; build-in type
05   Item     - object
06   Waypoint - object
07   Room     - Waypoint ; ...
08  )
09  (:constants
10   False - Boolean
11   True  - Boolean
12  )
13  (:predicates
14   (Item_name ?item - Item ?name - String) ; ...
15  )
16 )
```

Figure 8. Results from transforming the domain depicted in Fig. 2 to PDDL.

models, it creates intermediate data structures that represent a complete PDDL domain and several partial PDDL problems. To complement a PDDL problem, the planning infrastructure has to be additionally provided with an initial state and arguments for goal parameters. These are only available at runtime. Once the planner receives the information at runtime in form of Java objects that are instances of goal and property classes (*cf.* RTE in Fig. 7), the PDDL problem is produced and the PDDL planning problem can be solved. Thus, the PDDL domain is generated once at compile time and remains unchanged at run time. In contrast, for each individual runtime planning request, a new PDDL problem is produced. After solving a dynamically generated PDDL problem with respect to the domain generated once at compile time, the infrastructure transforms the PDDL planner's output back to a list of RTE class instances that represent sequences of actions. Under the assumption that the actions' symbolic preconditions and postconditions adequately reflect the actions' implementations, executing the actions in order leads to physical goal satisfaction.

A. Intra-Language Model-to-Model Transformations

The infrastructure generates a single PDDL domain from a CD domain model, a robot entity, and a world entity. To ensure the PDDL domain is well-formed, various transformations are applied to the entities. The first transformation ensures that no two actions have the same name. To this effect, the transformation prefixes each action name with the full-qualified name of its enclosing entity (using _ as package delimiter). The same is performed for properties. For instance, the name of the action `move` depicted in Fig. 4, is transformed to `TransportRobot_move`. To reduce notational complexity, the examples for the transformations following in the next sections do not include these preparation transformations.

a) From CDs to PDDL Types and Predicates: Each CD class is transformed to a PDDL type and each attribute of each class is transformed to a PDDL predicate. The generated PDDL types preserve the inheritance relation of the CDs and have the same names as the classes they originate from. For instance, the classes depicted in Fig. 2 are transformed to the PDDL types illustrated in Fig. 8 (ll. 5-7). The types `String` and `Boolean` (ll. 3-4) are built-in and may be used in domain and entity models. The name of the PDDL predicate derived from an attribute of a class is the name of the class concatenated to an underscore (_) and the name of the attribute. Each predicate has two parameters. The first parameter has the type of the class, whereas the second parameter has the type of the attribute. The class `Item` depicted in Fig. 2, for instance, has an attribute `name` of type `String`. Thus, the class, together with the attribute `name`, defines the binary PDDL predicate `(Item_name ?item - Item ?name - String)` depicted in Fig. 8 (l. 14).

b) From Entities to PDDL Predicates and Actions: Each property of each entity is transformed to a PDDL predicate. A property has parameters and a return value, whereas PDDL predicates represent Boolean predicates on a subset of the domain's types. With this, properties represent functions that are transformed to PDDL predicates. Therefore, a property with n arguments is transformed to an $(n{+}1)$-ary PDDL predicate. The first n predicate parameters have the PDDL types corresponding to the types of the parameters of the property. The last argument of the predicate has the PDDL type corresponding to the return value type of the property. Each PDDL predicate has the same name as its corresponding property. For instance, the `RoomsWorld` properties (Fig. 3, ll. 3-5) are transformed to the PDDL predicates `itemLoc` and `adjacent` (Fig. 9, ll. 3-4) and the `TransportRobot` properties (Fig. 4, ll. 4-5) are transformed to the PDDL predicates `robotLoc` and `hasLoaded` (Fig. 9, ll. 5-6).

```
01 (define (domain myDomain) ; ...                              PDDL
02  (:predicates
03   (itemLoc ?item - Item ?room - Room ?res - Boolean)
04   (adjacent ?w1 - Waypoint ?w2 - Waypoint ?res - Boolean)
05   (robotLoc ?res - Waypoint)
06   (hasLoaded ?item - Item ?res - Boolean) ; ...
07  ) ; ...
08 )
```

Figure 9. The properties of the entities depicted in Fig. 3 and Fig. 4 in PDDL.

Each action of each entity model is transformed to a PDDL action. The parameters and preconditions of entity actions correspond to parameters and preconditions of PDDL actions. The postconditions of entity actions correspond to PDDL effects. After transforming an entity's action, the resulting PDDL action yields the same name as the entity's action. The parameters of the resulting PDDL action yield the same names and types as the parameters of the entity's action.

c) From Action Preconditions to PDDL Preconditions: The precondition of an entity's action is a Boolean expression consisting of logical conjunctions, logical disjunctions, logical negations, and expressions referencing properties as well as attributes of the enclosing action's parameters. The transformation of action preconditions to PDDL preconditions preserves logical negations, conjunctions, and disjunctions. Expressions referencing values of properties as well as qualified expressions referencing attributes of domain model classes become corresponding PDDL expressions. In preconditions, values of attributes and properties may be queried and compared to parameters of the enclosing action, attributes of objects, or values requested from properties.

```
01 (:action move                                                    PDDL
02  :parameters (?from - Waypoint ?to - Waypoint)
03  :precondition (AND (robotLoc ?room)
04                     (adjacent ?from ?to True))
05  :effect: (AND (forall (?X_0 - Waypoint)
06                        (WHEN (AND (not (= ?X_0 ?to)))
07                              (AND (not (robotLoc ?X_0)))))
08                 (robotLoc ?to ))
```

Figure 10. The action `move` of the entity depicted in Fig. 4 in PDDL.

The result from transforming the precondition of action `move` (Fig. 4, ll. 6-9) is depicted in Fig. 10 (ll. 3-4). The intention is that the PDDL expressions `robotLoc ?room` and `adjacent ?from ?to True` are satisfied if, and only if, the property `robotLoc` returns `True` and the value of property `adjacent` applied to `?from` and `?to` yields `True`. The transformation of preconditions containing qualified expressions that reference values of domain model instances, *e.g.,* `i1.name == i2.name` where `i1` and `i2` are of type `Item` (Fig. 2 l. 3), is more involved. The resulting PDDL expression checks whether there is an object reachable by chaining the predicates introduced for the attributes referenced on both of the expression's sides. The example above, for instance, results in the following PDDL expression:

```
exists (X_0 - String X_1 - String)
 (AND (Item_name ?i1 X_0) (Item_name ?i2 X_1) (= X_0 X_1))
```

The intended PDDL precondition's meaning is the following: there exist x_0 and x_1 such that $x_0 = i1.name$, $x_1 = i2.name$, and $x_0 = x_1$. Expressions comparing the values of two properties are similarly transformed to PDDL.

d) From Action Postconditions to PDDL Effects: A postcondition of an entity action is a Boolean expression that consists of logical conjunctions and infix expressions, *i.e.,* expressions of the form `X == Y`, where `X` and `Y` are expression that either query the value of a property, a parameter, or a parameter's attribute. Simply specifying an expression `B` is syntactic sugar for `B == True` and `!B` is syntactic sugar for `B == False`. Each postcondition can be interpreted as a sequence of assignments, where each either assigns a new value (Right-hand side `Y`) to a property or to an attribute (left-hand side `X`). The transformation preserves logical conjunctions and transforms infix expressions to PDDL expressions for removing or adding facts.

The postcondition of the action `move` (Fig. 4, l. 8), for instance, assigns the value of the parameter `to` and the property `robotLoc()`. The corresponding generated PDDL effect (Fig. 10, ll. 5-8) ensures that the object encoded by `?to` is the only object that satisfies the predicate `robotLoc`. To this effect, for all objects `?X_0` (l. 6) such that `X_0`$\neq$`?to` (l. 6), the effect deletes the fact `(robotLoc ?X_0)` (l. 7) and then adds the fact `(robotLoc ?to)` (l. 8). The condition `X_0`$\neq$`?to` (l. 6) ensures the effect does not introduce an inconsistent planning state. Without the condition, the effect would first add the fact `(not (robotLoc ?to))` and afterwards add the fact (`robotLoc ?to`), which introduces a planning state inconsistency. Expressions used in postconditions that assign the value of a property to another property or assign the value of a property to an attribute, and vice versa, are transformed similarly as above.

B. From Goals to PDDL Problems

Each goal encodes a partial PDDL problem. In contrast to PDDL problems, goals can be parametrized. Hence, a goal only partially defines a PDDL problem. Additionally, they neither encode information about existing objects, nor about currently valid properties. In contrast, each PDDL problem defines a set of existing objects and an initial state consisting of facts relating objects (*i.e.,* the corresponding domain's predicates). Thus, PDDL problems cannot be generated at compile time, but only during run time when the current world and robot states are known, *i.e.,* it is known, which objects exist and, which properties are valid. Given a list of `Property` (Sec. IV) instances, which encode facts holding at a certain point in time, and a `Goal` (Sec. IV) instance, which assigns values to parameters of the corresponding goal, at run time, a well-formed and complete PDDL problem can be generated. The PDDL problem of Fig. 11, *e.g.,* is an excerpt of a PDDL problem generated from the goal model depicted in Fig. 5, a list of `Property` instances, and a `Goal` instance.

The PDDL problem generator takes a list of `Property` instances, a goal model, and a `Goal` instance corresponding to the goal model and generates a PDDL problem as follows:
(1) Each attribute of each `Property` instance becomes an object in the PDDL problem (ll. 2-6). The identifier of each object is determined by concatenating the parts of the full qualified class name of the object before suffixing the object's hash-code to the result of the concatenation.
(2) Each `Property` instance becomes an initial fact in the PDDL problem (ll. 7-13). The name of the fact is the full qualified name of the class of the `Property` instance. Each attribute of the `Property` instance becomes a parameter of the fact. The parameters' identifiers are determined as in (1).
(3) The goal formula of the PDDL problem (ll. 14-19) is generated from the predicate of the goal model, and the attributes of the `Goal` instance. The attributes of the `Goal` instance instantiate the parameters of the goal model. Therefore, the goal of the PDDL problem results from applying the following

```
01 (define (problem LoadedAt)                                    PDDL
02  (:objects
03   logisticsdomainItem141 - Item
04   logisticsdomainItem149 - Item
05   logisticsdomainRoom412 - Room ; ...
06  )
07  (:init
08   (itemLoc  logisticsdomainItem141
09             logisticsdomainRoom412 True)
10   (itemLoc  logisticsdomainItem149
11             logisticsdomainRoom412 True)
12   (robotLoc logisticsdomainRoom412 True) ; ...
13  )
14  (:goal ; Derived from goal "loadedAt"
15   (exists (?room - Room ?item - Item)
16           (and
17            (= ?room logisticsdomainRoom412)
18            (= ?item logisticsdomainItem149)
19            (and (robotLoc ?room) (hasLoaded ?item True))))
20  )
21 )
```

Figure 11. PDDL problem derived from the goal depicted in Fig. 5 and instances of the classes generated from the properties of the entities depicted in Fig. 3 and Fig. 4 as well as the domain depicted in Fig. 2.

two transformation steps: First, the predicate given in the goal model is transformed the same way as preconditions of actions are transformed to PDDL preconditions (l. 19). Afterwards, the result of the former transformation is enclosed by a term that binds the goal's parameters by existential quantification (l. 15) and requires equality to the values defined by the attributes of the `Goal` instance (ll. 17-18).

The result from solving a PDDL problem with respect to a PDDL domain is a sequence of actions defined in the domain where the actions' parameters are bound to objects defined in the PDDL problem. After planning, the reference architecture maps each action of the plan back to its entity action and each PDDL object back to the instance it originated from. With this information, for each action in the plan, the architecture creates an instance of the class generated from the entity's action and sets the instance's attributes to the domain instances corresponding to the objects bound to the action in the plan. The architecture then adds the action instances to a list (representing a sequence of actions) in the order defined by the plan and forwards it to trigger their execution.

VI. Related Work

Modeling techniques for robotics focus on abstraction in the solution domain [10]. Techniques for the problem domain are rare and usually tied to specific platforms. The textual DSL for modeling robot abilities presented in [11], *e.g.,* also enables defining sequences of actions that are independent from specific robots. It is implemented as a DSL embedded in Java and requires imperative programming of tasks. Fixing tasks this way eliminates the flexibility of on-line planning in cases where the environment at system run time differs from assumptions made at system development time. The ontology for service robot behavior presented in [12] resembles the DSLs we presented, but distinguishes sensing actions from manipulation actions. The authors present common control structures for robotics applications, whereas we did not consider confronting the domain experts with control structures in the tasks. The concepts presented in the ontology are not realized as a modeling technique. Various robotics specific modeling techniques defined on top of the situation calculus [13] enable describing robotic goals and actions and properties required to fulfill these in a platform-independent fashion. These DSLs are tailored to knowledge representation experts.

VII. Discussion

The iserveU DSLs enable separating the concerns of application domain experts from robotics experts. Consequently, the DSLs are designed to be uncomplicated (*i.e.,* there are no conditionals or loops). However, formulating goals still requires understanding the dot-notation typical to object-oriented programming and describing Boolean expressions. Comprehending dot-notation and Boolean expressions requires a certain skill set from the domain expert. Whether other notations are better suited for this is subject of research. We also did not include heuristics into planning. Despite being able of accelerating planning, defining heuristics requires modeling relevant properties of tasks and goals as well as heuristic functions the planner can optimize. With the aim of providing straightforward DSLs, we refrained from that. Our approach requires to explicitly model all effects of an action that influence the world state. If this is incomplete, a plan may not be executable, because the models rely on an erroneous description of the current situation. Further, describing non-functional properties as, *e.g.,* the robot has to arrive at a certain location within a given amount of time, is impossible or complicated with the described approach. Decoupling the robot's actions from their problem domain representation enables exchanging the underlying platform easily: this requires implementing the interface generated from the robot model only. Hence, the same domain model, tasks, goals, and model transformations can be reused to execute robot actions with different robots with little effort.

VIII. Conclusion

We presented a family of executable DSLs that support describing the concepts of service robotics applications in a platform-independent fashion by separating the different stakeholders' concerns. Models of these languages are transformed into parts of the execution framework at design time as well as into PDDL problems at system run time. Based on these problems, the integrated Metric-FF planner computes a series of actions that ultimately are executed using loose bindings to the underlying robot platform. Execution via transformation to an established planner yields the benefit of run-time flexibility required for dynamic real-world environments.

References

[1] H. J. Levesque, R. Reiter, Y. Lesperance, F. Lin, and R. B. Scherl, "GOLOG: A Logic Programming Language for Dynamic Domains," *Journal of Logic Programming*, 1997.

[2] R. Heim, P. Mir Seyed Nazari, J. O. Ringert, B. Rumpe, and A. Wortmann, "Modeling Robot and World Interfaces for Reusable Tasks," in *Intelligent Robots and Systems Conference (IROS'15)*, 2015.

[3] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng, "ROS: an open-source Robot Operating System," in *ICRA Workshop on Open Source Software*, 2009.

[4] K. Adam, A. Butting, R. Heim, O. Kautz, B. Rumpe, and A. Wortmann, “Model-Driven Separation of Concerns for Service Robotics,” in *International Workshop on Domain-Specific Modeling (DSM’16)*, 2016.

[5] B. Rumpe, *Modeling with UML: Language, Concepts, Methods*. Springer International, July 2016.

[6] R. E. Fikes and N. J. Nilsson, “STRIPS: A new approach to the application of theorem proving to problem solving,” *Artificial intelligence*, vol. 2, 1971.

[7] J. O. Ringert, A. Roth, B. Rumpe, and A. Wortmann, “Language and Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems,” *Journal of Software Engineering for Robotics (JOSER)*, 2015.

[8] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins, “PDDL-The Planning Domain Definition Language,” Yale Center for Computational Vision and Control, Tech Report, 1998.

[9] J. Hoffmann, “The metric-FF Planning System: Translating "Ignoring Delete Lists" to Numeric State Variables,” *Journal of Artificial Intelligence Research*, 2003.

[10] A. Nordmann, N. Hochgeschwender, D. L. Wigand, and S. Wrede, “A survey on domain-specific modeling and languages in robotics,” *Journal of Software Engineering in Robotics*, 2016.

[11] M. Reckhaus, N. Hochgeschwender, P. G. Ploeger, G. K. Kraetzschmar, and S. Augustin, “A Platform-independent Programming Environment for Robot Control,” in *Proceedings of the 1st International Workshop on Domain-Specific Languages and models for Robotic systems*, 2010.

[12] J. P. Diprose, B. Plimmer, B. A. MacDonald, and J. G. Hosking, “How People Naturally Describe Robot Behaviour,” in *Proceedings of Australasian Conference on Robotics and Automation*, 2012, pp. 3–5.

[13] J. McCarthy and P. J. Hayes, “Some Philosophical Problems from the Standpoint of Artificial Intelligence,” *Machine Intelligence*, 1969.