The AI Domain Definition Language (AIDDL) — Three Use Cases for Integrated Planning, Reasoning, and Learning

Uwe Köckemann

Center for Applied Autonomous Sensor Systems
Örebro University, Sweden uwe.kockemann@oru.se

Abstract

As individual sub-fields of AI become more developed, it becomes increasingly important to study their integration into complex systems. In this paper, we look at three examples of how automated planning can be integrated with learning and reasoning. We then provide a first look at the *AI Domain Definition Language (AIDDL)* as an attempt to provide a common ground for modeling problems, data, solutions, and their integration across all branches of AI in a common language.

Introduction

Many possibilities arise when combining the strengths of different AI methods. Automated planning, for instance, has been combined with machine learning or reasoning in various ways for mutual benefit(Coles and Coles 2007; Gerevini, Saetti, and Vallati 2009; Jiménez et al. 2012). In this work, we present the *AI Domain Definition Language (AIDDL)* and framework (available under aiddl.org), which aim at allowing AI system developers to easily integrate different AI models. AIDDL is flexible, extendable, and usable for a any type of AI problem because new types can be defined within the language itself. It also allows to easily exchange algorithms and solutions for individual sub-problem, which make studying alternative combinations of solution easy. It also facilitates the upgrade of the whole system to latest state of the art advances. Finally, by using AIDDL, the integration of the different AI models can be moved from implementation- to model level. This means that integrated AI systems can be described independent of any programming language. Thus, creating and maintaining AI systems becomes easier as the developer can model interactions between the system's component without having to worry about implementation details.

In this paper, we will consider three examples of integration of different AI models, namely automated planning, learning and reasoning. These examples will be integrated within the AIDDL framework which is composed of the *AI Domain Definition Language (AIDDL)*, a core library (AIDDL Core), definitions and implementations of common AI algorithms (AIDDL Common), and a library of examples of integrated AI (AIDDL Examples).

The remainder of this paper is organized as follows. First, we discuss three examples of integrative AI that combine planning with learning or reasoning. After discussing these examples, we provide an overview of the current version of the *AI Domain Definition Language (AIDDL)* to model problems and solutions for integrative AI. We discuss our design goals, the language format, and the components of the framework: AIDDL Core, AIDDL Common, and AIDDL Examples. We then take a detailed look at how one of the three presented examples is realized. Finally, we discuss related work and directions for future work.

Background

This paper focuses on examples integration of classical planning, supervised learning, and goal reasoning. Therefore, this section will provide a short introduction to these three concepts to allow the reader to fully understand the examples and the notations used through this paper. It is important to note however that AIDDL is not limited to these models but can be used (and extended) to any other type of algorithms and AI problem.

A classical task planning problem (Ghallab, Nau, and Traverso 2004) (S_0, G, O) consists of an initial state S_0, a goal state G and a set of operators O. Both S_0 and G are state-variable assignments. An operator $o = (n_o, P_o, E_o)$ consists of a name n_o, set preconditions P_o and effects E_o. Both, P_o and E_o are state-variable assignments. A goal or precondition G is satisfied in a state S if $G \subseteq S$. Applying an effect E to a state S replaces the value assignments of all state-variables in S with the ones assigned by E. A plan $\pi = a_1, \ldots, a_n$ is a sequence of operators that can be applied starting at the initial state S_0 leading to the state sequence $S_0, S_1, \ldots, S_n$. A plan is applicable if $\forall_i P_{a_i} \subseteq S_{i-1}$. A plan is a solution if it results in state S_n with $G \subseteq S_n$. Our planner uses forward-state space search with the Fast Forward heuristic(Hoffmann 2001).

A supervised machine learning problem (Mitchell 1997) consists of a data set $D = \{d_1, \ldots, d_n\}$ where each $d_i = (l_i, x_i)$ consists of a label $l_i \in L$ and a vector $x_i \in X$. The goal is to learn a mapping $f : X \rightarrow L$ that accurately predicts the label l of any point in X. There are many variations of machine learning problems depending on the domains X and L. In this paper we use finite symbolic domains for L and vectors of finite symbolic domains for X in both examples. As a learner we use a very basic approach to decision tree learning called ID3(Mitchell 1997, Ch.3). A common way to test the model produced by a machine learning algorithm is cross validation (Murphy 2012, p.24). The idea of cross validation is to divide that data into n partitions (folds), and always use one fold for testing and all others for training.

The goal reasoning method used in the third case study below looks at sub-class relations between objects. These relations are expressed in a directed acyclic graph. A path in this graph indicates that an object is a concrete instance of a certain class. Therefore, this simple reasoning problem can be solved by applying a path finding algorithm from an object used in a goal to an object known to the planner. This way of reasoning could easily be replaced by more general reasoning methods. Prolog(Bratko 2000), for instance, is a declarative language to model relational knowledge in predicate logic that could be use to express a goal hierarchy and answer queries about sub-class relationships. Our third case study has one solution using a simple graph search and one using a wrapper for YAP Prolog (Costa, Rocha, and Damas 2012).

Three Case Studies

All of the following examples use a similar style of flow chart (see Figures 1 through 3). Yellow square boxes are the function calls. Some of these refer to basic AI functionality (e.g., Plan, ID3) that appears in multiple or all our examples. Other functions (e.g., Generate Data Goals) are specific to the aim of the example integration. Blue diamond shaped nodes are decision points. Orange hexagon shaped nodes indicate loops. The inner part of each loop is highlighted by a bounding box. Finally, green cylinder shaped nodes indicate data (data was omitted in our first two examples to avoid cluttering the figures). All flow charts were automatically created from their AIDDL request models and modified by hand to increase clarity.

Planning for Learning

Data is often difficult to acquire and data acquisition may require going through a complex process such as sending a robot to collect samples. In this section we show a basic approach that allows using an automated planner to gather data for a machine learning system.

The flow diagram in Figure 1 shows the overall process. Until our model performs well enough on a cross-validation, we generate data goals, plan to collect data, execute the plan, extract data, and then perform n-fold cross-validation.

Ideally, we would like to be able to integrate planning and learning in this way to be as simple as creating this flow chart. In fact, the only non-standard components in this example are *Generate Data Goals* and *Extract Samples*.

Learning for Planning

In this example we use learning for planning. Specifically, we start with an incomplete planning domain, generate data from executed actions and observed state transitions, and then learn new operators. If the problem cannot be solved

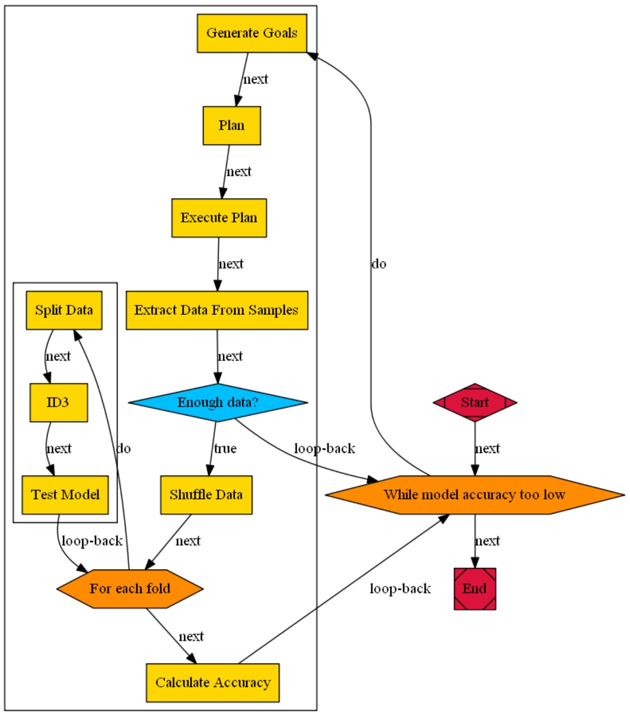

Figure 1: Using a planner to collect data for a learning system.

a random action is chosen as an experiment. Otherwise we execute the next action of the plan. In either case, we execute the action, observe the outcome and generate data. Next, we try to learn a model that predicts the effects of actions on states. The resulting model is used to create operators for an updated planning domain. Once the domain is updated, we attempt to plan again and continue.

As before, we have components related to learning, planning, and execution and a few non-standard boxes (*Experiment*, *Extract Data*, *Generate Operators*) that need to be defined to implement the integration. Considering the first example, we would like to re-use as many components as possible, and then simply decide how to choose experimental actions, how to extract data, and how to create operators from a learned model.

Apart from making it easier to create integrative AI systems, this view also enables an convenient way to perform experiments on/with such systems. There are many ways to decide what exactly the *Experiment* box in Figure 2 does. It may choose an action at random, consider its previous choices, or consider how well the current set of operators performs. A similar point can be made for the *Generate Data Goals* box in Figure 1.

Planning and Goal Reasoning

Our third example integrates automated planning with a form of goal reasoning. Here, we consider a planner that may be presented with a goal that it does not support directly. In case this happens, we call a reasoner to derive a

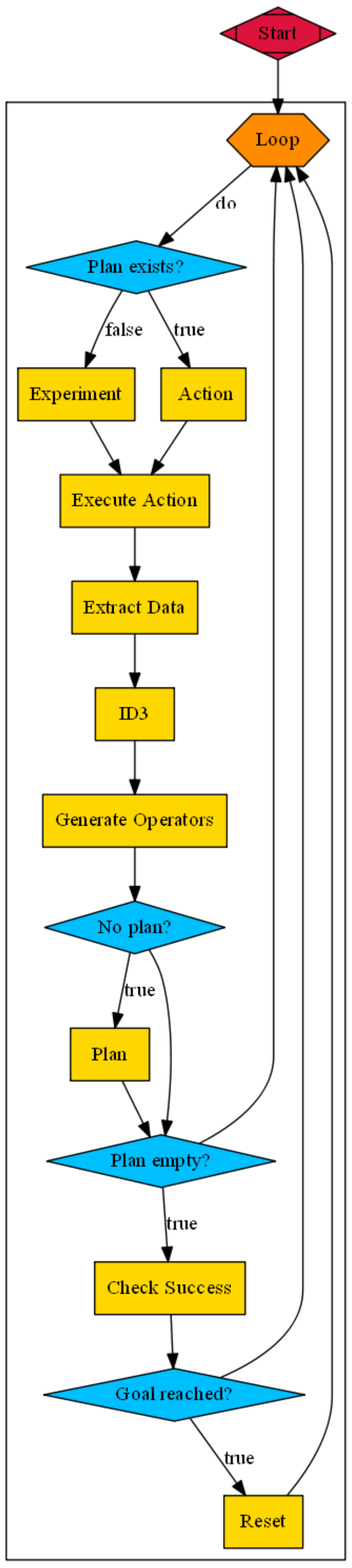

Figure 2: Using a learner to acquire domain knowledge for planning.

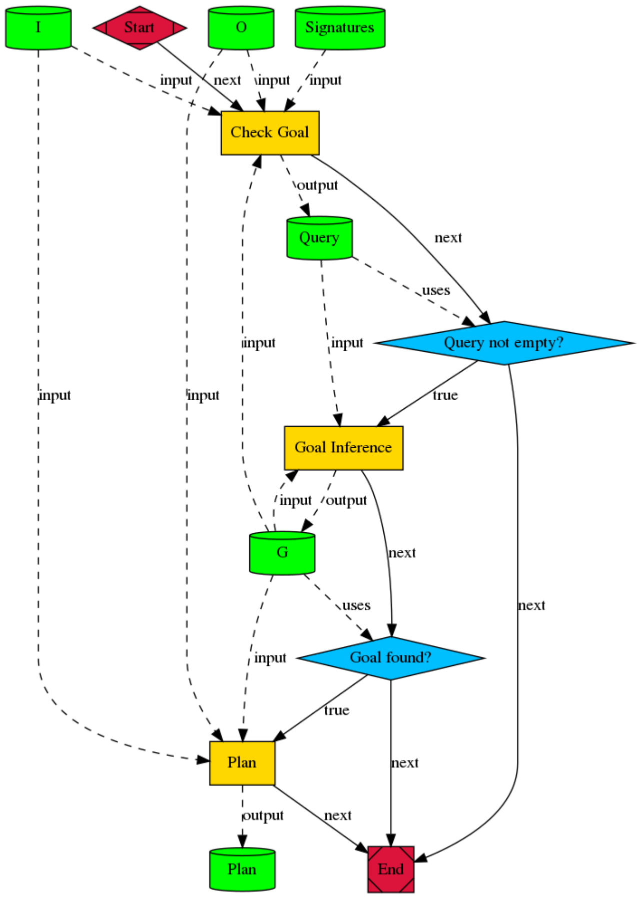

Figure 3: Performing goal reasoning to transform goals for a planner.

goal that the planner can handle. As an example, consider that a planner may have a set of locations $\{l_1, \dots, l_n\}$ and is presented with a goal to go to $kitchen$. We have a knowledge base of sub-class relations and instance relations and want to determine if any of the l_i locations known to the planner is a $kitchen$. Figure 3 outlines how our AI components are connected. Here we also indicate the data flow between the different components.

Apart from the fact that these AI components can be easily exchanged, this simple integration enriches the types of goals that a planner can handle without touching the planner itself. While our example of sub-class relations is trivial, it is easy to see how more complex goal reasoning may be performed. A goal such as *"buy food"* may be decomposed to create a shopping list based on a meal plan and an overview of groceries still in storage.

Planning and reasoning could benefit from each other in many other ways. For instance, instead of reasoning on the goal, we could extend the initial state by deriving further facts from background knowledge. Another approach could use the set of goals to task a reasoning system to focus the domain of the planner on a subset of operators and state factors to reduce the search space of the planner task. Ideally, these examples should boil down to re-arrangements of Figure 3.

The AIDDL Framework for Integrated AI Systems

The previous examples are different in scope but they all share at least the planner as a common component. In this work, we propose the *AI Domain Definition Language (AIDDL)* as a representation to make these examples of integration easy task. Ideally, integration should become as easy as creating one of the presented flow charts, add some missing components and execute the solution. The AIDDL framework is a step in this direction. Our main design goals are:

- *Well-defined Integration*: allow to create models of how AI algorithms are connected (possibly reasoning about these connections)
- *Simplicity*: simple tasks require simple models
- *Emerging Complexity*: connect simple and well-tested pieces to build complex AI systems
- *Re-usability*: abstract away common patterns into type definitions and software modules
- *Robustness*: types and implemented functionalities can be tested individually
- *Freedom*: not imposing a specific programming language or way of thinking or way of problem solving

The AIDDL framework is composed of the language (AIDDL), as well as the Core, Common, and Example libraries. An overview of the framework can be found in Figure 4. *AIDDL* proper specifies types, data, functionalities, and requests. Consider the following examples:

- **Types:** Planning Problem, Plan, Decision Tree, Supervised Learning Problem

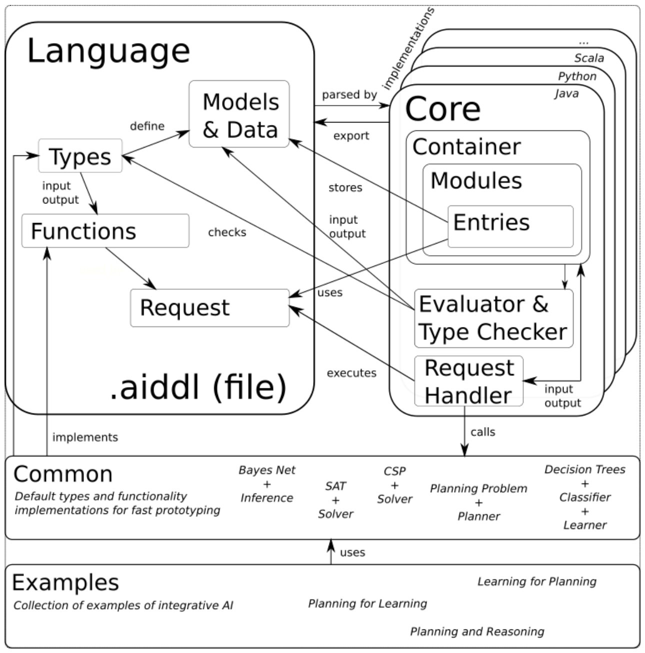

Figure 4: An overview of the AIDDL framework.

- **Model/Data:** Instance of planning problem, instance of decision tree, sensor readings, etc.
- **Functionality:** Planner takes instance of planning problem and creates a plan
- **Request:** Figures 2 through 3 are visualizations of complex requests represented in AIDDL.

The *Core* parses and exports AIDDL files, handles requests and is capable of evaluating expressions, as well as type checking. *Common* contains AIDDL modules defining types of common AI problem and implementations of solvers for these problems. Finally, *Examples* is a collection of example integrations containing the three examples discussed above. Both *Common* and *Examples* contain implementations of functionalities (i.e. the yellow square nodes in the example figures). There are two additional libraries not further detailed in this paper. *Util* contains additional functionality. Most notably it allows to host and use the AIDDL framework via Protobuf and gRPC[1] which allows to distribute computing. Finally, *External* will contain various libraries whose purpose is the integration with other AI frameworks. Integration with *Scikit Learn*[2] and the *Unified Planning* framework[3] are currently under development.

AIDDL, as specified by the grammar below, allows to define data types and data for AI problems. It also allows to connect functions that solve these AI problems to each other. As a result, all three examples presented above can be expressed fully in AIDDL. Each AIDDL file represents a *module* and conforms to the following grammar.

[1]https://www.grpc.io/
[2]https://scikit-learn.org/
[3]https://www.ai4europe.eu/research/ai-catalog/unified-planning-framework

```
<AiddlFile> :: <Module> (<Entry>)*
<Module>    :: "(#mod" <Symbolic>
                        <Symbolic> ")"
<Entry>     :: "("<Term> <Term> <Term>")"
<Term>      ::  <Basic> | <Composite>
             |  <Reference>
<Basic>     :: <Symbol> | <Numerical>
             | <Variable> | <String>
<Numerical> :: <Integer> | <Rational>
             | <Real> | <Infinity> | <NaN>
<Symbol>    :: (("a"-"z"|"A"-"Z"|"#")
                ("a"-"z"|"A"-"Z"|"0"-"9"
                |"_"|"."|"-"|"′")*)
                |"+"|"-"|"/"|"*"|"&"|"|"
                |"!"|"="|"<"|">"|"=>"
                |"<=>"|"^"|"!="|"<="|">="
<String>    :: "\"" [~\"]* "\""
<Variable>  :: <NamedVar> | "_"
<NamedVar>  :: ?(("a"-"z"|"A"-"Z")
                 ("a"-"z"|"A"-"Z"
                 |"0"-"9"|"_"|"."|"-"|"′")*)
<Integer>   :: ["-"]("0"|"1"-"9")("0"-"9"]*
<Rational>  :: ["-"]("0"|"1"-"9")("0"-"9")*
               "/" ("1"-"9"("0"-"9")*)
<Real>      :: ["-"] ("0"|"1"-"9")("0"-"9")*
                "." ("0"-"9")+
<Infinity>  :: ["+"|"-"]"INF"
<NaN>       :: "NaN"
<Composite> :: <List> | <Set>
             | <Tuple> | <KeyValue>
<List>      ::  "[" <Term>* "]"
<Set>       ::  "{" <Term>* "}"
<Tuple>     ::  "(" <Term>* ")"
<KeyValue>  :: [<Basic>|<List>|<Set>
               |<Tuple>
                |<Reference>]":"<Term>
<Reference> :: <EntRef> | <FunRef>
<EntRef>    :: [<Symbolic>|<Tuple>]
               "@"<Symbolic>
             | "$"<Term>
<FunRef>    :: "^"[<Symbolic>|"$"<Symbolic>
                  |<Symbolic>"@"<Symbolic>]
```

Each AIDDL file starts with a module entry which states the self-reference (i.e., the term a module uses to refer to itself) and the module's URI. An *entry* generically specifies data as a tuple composed of type, name, and value.

Requirement entries (type *#req*) state dependency on another module. The name of a requirement can be used locally to refer to its module. Namespace entries (type *#nms*) are used to replace each occurrence of a term with another. This can be used to avoid repeated usage of long names. Definitions (type *#def*) are used to define functions. The value of a definition is a term that can be evaluated. Type entries (type *#type*) define types as subsets of the language. Function interfaces (type *#interface*) consist of a URI, input- and output types for computations. An interface implementation can be linked to an *#interface* entry to allow the type checker to verify that input and output data satisfy the required type. Finally, request entries[4] (type *Request@org.aiddl.request*) can be used to specify control flow by linking service calls to entries in the AIDDL container.

AIDDL Core

The *Core* is a library that makes AIDDL available for use with a programming language. It provides a parser to load AIDDL files as modules into an AIDDL container. It also includes an implementation of the evaluator (also used as type checker) and request handler. Input for evaluator and request handler are described in the next two sections. All examples contained in this paper are implemented based on a Java version of the AIDDL Core. The Core is also available in the Python and Scala languages.

Evaluator

The evaluator is a component of the AIDDL Core that recursively evaluates terms that can be interpreted as functions. All such terms are tuples and the first term of the tuple is a symbolic URI of the function. As an example, *(org.aiddl.eval.numerical.add 2 3)* is evaluated to 5. The evaluator can be used to test if the value of an entry satisfies the stated type. It is also used to perform basic operations on AIDDL data, e.g., to evaluate branching or loop conditions in requests or filtering lists or sets without having to implement a functionality. All conditional branches and while-loops in the presented examples use the evaluator to test their conditions. Namespaces are used to shorten overly long names. One of the default namespaces allows writing the above example as *(+ 2 3)*. Default functions include basic operations on sets, lists, numerical values, logical operations (including quantification).

Requests

Requests compose functionality in an imperative fashion. Unlike the strictly functional interpretation of the evaluator, requests require to specify where the result of function calls is stored (i.e., the name of entries to direct the output to). Any evaluator term that appears in a request will be evaluated (as explained above) before the request is handled. This is often convenient for evaluating conditions. All examples presented in this paper were implemented using requests. The following list provides a short overview of requests with examples.

- Call functionality on input and write result to output.
  ```
  (call plan (s0 g O) pi)
  ```
- Execute requests in a sequence.
  ```
  [R1 R2 R3]
  ```
- Execute request while condition is true.
  ```
  (while (< acc 0.95) plan-and-learn)
  ```
- For each combination of variable-value pairs, create a substitution σ and call $R\sigma$.
  ```
  (forall [?x:[a b] ?y:[1 2]] (call f (?x ?y
    ) o))
  ```

[4]The name of this entry is a reference to the entry *Request* in module *org.aiddl.request*. The name is different because requests are defined in AIDDL as any other type.

- Loop indefinitely
  ```
  (loop [sense plan act])
  ```
- Match two terms and execute request on the resulting substitution if possible.
  ```
  (match (?x ?y) ((1 2) (3 4))(call dist
      (?x ?y) d))
  ```
- If condition holds execute first request, otherwise execute second request. (optional)
  ```
  (if (x < 10) R1 R2)
  ```
- Misc: write value to entry, create entry, print message, keep time.

AIDDL Common

The *Common* library consists of two elements: a library of type and functionality definitions, and a library of implementations. Type definitions are written in AIDDL and cover common AI problems and data types. Functionalities are defined for solvers of these common problems. Second, *Common* provides a library of implementations of AI functionality and some commonly used data structures (such as graphs and matrices). These can be used as building blocks for setting up and testing integrated AI systems.

Currently covered functions and types include: graphs (e.g., strongly connected components, shortest path). State-variable Planning with types for states, goals, operators, plan, relaxed planning graphs, domain transition graph, causal graph and functions for forward search, the fast forward heuristic (Hoffmann 2001), the causal graph heuristic (Helmert 2006). Machine Learning functions include the ID3 algorithm used above, a decision tree classifier, and various functions for testing. Reasoning functions include a basic SAT solver and conversion of first-order queries to Prolog. Temporal Reasoning is implemented by a solver for simple temporal problems (Cesta, A. and Oddi, A. 1996). Scheduling with reusable resource is implemented as a meta-CSP reasoner(Cesta, Oddi, and Smith 2002). Constraint processing by a basic combination of tree search and propagation (Dechter 2003).

AIDDL Examples

Examples refers to a growing collection of implementations of integrated AI. All three case studies presented in this paper are implemented as part of AIDDL Examples. Most of the functionalities used by these examples are implemented in AIDDL Common. This section describes the AIDDL version of the learning for planning example. The other two examples are not included due to space limitations. All three realizations of the presented case studies are available open source at aiddl.org.

The aim of this integration example is to create operators from observed state transitions. We use a state transition system where a state contains the status of 5 lights (true or false). We have 5 actions that correspond to buttons. Each action changes the state in an unknown way. There is one special action *reset* that just reverts the current state to the initial state to avoid dead-ends.

Terms such as *a*@*m* refer to the entry named *a* in module *m*. This is mostly used to reference types in other modules. Any term of the form *$ a* refers to an entry with name *a* in the module where it appears. Functions are referred to with the ^ symbol. If a function reference is combined with an entry reference (such as $\hat{f}$@*m* or $\hat{\$f}$), the entire expression is understood as a reference to a function in the specified module.

The following shows how *State*, *Goal*, and *Operator* can be defined. A state-variable assignment is a set consisting of key-value pairs $?K : ?V$ such that key and values are first-order logic atoms (defined in module *FL*), variables, or symbols. In a set of state-variable assignments, each key is only assigned one value (this is asserted by the attached constraint). States and goals are defined as state-variable assignments. An operator is a dictionary with a name of type *Atom@FL*, and preconditions, and effects of type *SVAs*.

```
(#req FL reasoning.logic.first-order)
(#req EVAL org.aiddl.eval.namespace)
(#nms nms-all all-ops@EVAL)

(#type E (union
  {^Atom@FL ^symbolic ^variable}))

(#type StateVariableAssignment
  (typed-kvp ^$E:^$E))

(#type SVAs
  (set-of ^$StateVariableAssignment
   constraint:(lambda ?X
                (is-unique-map ?X))))

(#type State ^$SVAs)
(#type Goal  ^$SVAs)

(#type Operator
  (dict
  [name:^Atom@FL
   preconditions:^$SVAs
   effects:^$SVAs]))
```

Below we show entries assigning initial state, goal, etc. Here the module CP contains our classical planning definitions (included with *# req*). Note the set of operators *O* and actions *A* are initially empty. Operators are added as domain knowledge is acquired, while actions are added initially when the state-transition system is randomly created (see *Init* below). Initially we have an empty set of operators and no plan.

```
(State@P s0    {(light 1) : false
                (light 2) : false
                (light 3) : false
                (light 4) : false
                (light 5) : false })
(State@P s      {})
(State@P s_next {})
(Goal@P  g      { (light 1) : true
                  (light 2) : true
                  (light 3) : true
                  (light 4) : true
                  (light 5) : true})
```

```
(Operators@P O {})
(set A {})
(Plan@P      pi NIL)
(Action@P    selected_action NIL)
(Problem@P Pi (operators:$O
               initial-state:$s
               goal:$g))
(boolean goal-reached false)
```

The next listing introduces the learning problem. First, we define the attributes of the data we are collecting. These include the current state and action taken, as well as observed effects. Our data set is initially empty. The machine learning problem is composed of attributes, label, and data.

```
(#req ML learning.supervised)
(#req DL learning.decision-tree)
(Attributes@ML PlanAttributes
  [ (light 1) (light 2) (light 3)
    (light 4) (light 5) Action Effects])
(DataPoints@ML Data {})
(Problem@ML MLProblem
  (attributes : $PlanAttributes
   label      : Effects
   data       : $Data ))
(DecisionTree@DL DT [])
```

Now we compose an integrated AI method that uses the data specified above. For readability, we divide the actual request into four parts defined below. The overall system runs in an infinite loop and does not change initial state or goal. In a more realistic setting, there should be some measure of how well the current operators reflect the state transition system and some variation of initial state and goal.

```
(#req R org.aiddl.request)
(Request@R Main [ $Init
  (loop [
    $SelectAndExecuteAction
    $ExtractAndLearn
    $PlanAndCheckSuccess
   ])])
```

Some of the calls below have complex inputs and outputs (splitting their outputs to multiple entries). *Init* sets the current state *s* to the initial state and create a random state transition system, which will decide what actions do in any given state.

```
(Request@R Init [
  (write $s0 s)
  (call sim-generator $N
    {state-transitions:Sigma
     actions:A})])
```

SelectAndExecuteAction either selects the next action of the current plan if it exists or a random action as an experiment. Note that *select-action* here contains both cases of the conditional in Figure 2. Otherwise a random action will be selected as an "experiment".

```
(Request@R SelectAndExecuteAction
  (call select-action
        (actions:$A plan:$pi
         state:$s sigma:$Sigma)
        {selected-action:selected_action
         plan-tail:pi})
  (call execute-action
        (state:$s action:$selected_action
         sigma:$Sigma) s_next))
```

ExtractAndLearn extends our data set with the current state transition and action, then uses the ID3 decision tree learning algorithm on the data and generates operators from the resulting decision tree.

```
(Request@R ExtractAndLearn
  (if (!= $selected_action (reset)) [
    (call extract-data
      (attributes:$PlanAttributes state:$s
       next-state:$s_next
       data:$Data
       action:$selected_action) Data)
    (call ID3 $MLProblem DT)
    (call generate-operators $DT O) ]) )
```

Finally, *PlanAndCheckSuccess* advances the current state, attempts planning if no plan exists, and checks if the goal was reached if the current plan is empty (i.e., it has been fully executed). If we reach the goal, we revert back to the initial state.

```
(Request@R PlanAndCheckSuccess [
  (write $s_next s)
  (if (= $pi NIL) (call plan $Pi pi))
  (if (= $pi [])
  [
    (call check-success
      (state:$s goal:$g) goal-reached)
    (if $goal-reached
      [(write $s0 s)
       (write NIL plan) ])]])
```

Related Work

Our work is motivated by work on domain definition languages for dedicated purposes (such as planning) and by many approaches that realize systems often very similar to our case studies. The AIDDL framework can be seen as an attempt to connect the kind of works listed in the following two paragraphs without creating yet another dedicated domain definition language.

Research on automated planning has lead to many changes and variations of the Planning Domain Definition Language (PDDL)(Ghallab et al. 1998; Fox and Long 2003; Gerevini and Long 2005; Coles and Coles. 2014) that to consider, e.g., time, resources, or continuous change. The basic language, however, is designed to express planning problems. In this work we take a step back and suggest a domain definition language that can be extended to include any existing AI problem and also define how these problems are solved. Unlike languages such as Prolog, we do not assume anything about how problems are solved.

The examples we look at in this paper are somewhat simplified takes on integrative AI. Learning and planning have been combined with many different motivations. Our first example uses learning to acquire domain knowledge. Early approaches of this motivation for integration can be found in (Shen and Simon 1989), (Gil 1992), and (Wang 1994). For a more recent review of this line of research, see (Jiménez et al. 2012). Our second example can be seen as an extension of active learning (Tong 2001). Active learning uses queries to ask for specific instances of data to improve the performance of learning. In our case, a query of the active learner is a data goal which is satisfied by a planner. The data is provided as a result of plan execution. *Generate Data Goals* (see Figure 1) can be seen as an active learning module. Other ways of integrating planning and learning include learning heuristics for the search space of a planner(Thayer, Dionne, and Ruml 2011), learning control knowledge to guide planning search(Coles and Coles 2007), or configuring portfolio-based planning approaches(Gerevini, Saetti, and Vallati 2009). Our third example can be seen as a subproblem of goal reasoning (Vattam et al. 2013) in form of goal transformation (Cox and Dannenhauer 2016).

There are many forms of integrative AI that do not include planning. Statistical relational learning combines first-order logic, probabilistic reasoning, and learning(Getoor and Taskbar 2007). Satisfiability Modulo Theories (SMTs) (Barrett et al. 2009) include a correspondence between propositional logic and models of other types of knowledge (such as linear equations). The SMT approach has been adopted to planning as well in form or Planning Modulo Theories (Gregory et al. 2012). In this style of integration every modulo theory T increases the expressiveness of operators. Theories have to be evaluated to determine applicability and when applying actions. We hope to extract patterns for integrative AI systems from these and similar approaches in future work.

Conclusion

We discussed a series of ways to integrate planning with learning and reasoning and proposed the AIDDL Framework for integrative AI. This way of performing AI integration allows the resulting systems to benefit from a large body of existing implementations across all branches of AI. We also argue that a common domain definition language for AI allows to perform integration as done in this paper with minimal overhead.

Notably all of the examples presented in this paper use planning and learning methods in form of a black box. Many existing approaches for integrative AI use a white-box view where an algorithm is modified or extended in some way to accommodate some other AI technique. Currently, we are investigating such white-box approaches to integration within AIDDL in which we compose components of a planner (such as search space expansion, heuristic computation) with reasoning about time or resources.

In addition, we would like to implement forms of integration that do not include automated planning to test the suitability of AIDDL for general purpose AI integration. We are quite positive about this outlook since none of the features of AIDDL is inherently linked to automated planning.

If you are reading this paper and made it this far, we hope we spiked your curiosity. Feel free to contact us if you have any further questions, a use case, or would like to hook up your AI tools to the *AIDDL Framework* available under aiddl.org.

Acknowledgement. This work has been supported by the European Union's Horizon 2020 research and innovation programme under grant agreement No. 101016442 (AI-Plan4EU).

References

Barrett, C.; Sebastiani, R.; Seshia, S. A.; and Tinelli, C. 2009. Satisfiability modulo theories. In Armin Biere Marijn Heule, H. v. M. T. W., ed., *Handbook of Satisfiability*, volume 185, chapter 26, 825–885. Frontiers in Artificial Intelligence and Applications.

Bratko, I. 2000. *Prolog Programming for Artificial Intelligence*. Addison Wesley. ISBN 0201403757.

Cesta, A.; Oddi, A.; and Smith, S. F. 2002. A Constraint-Based Method for Project Scheduling with Time Windows. *Journal of Heuristics*, 8(1): 109–136.

Cesta, A. and Oddi, A. 1996. Gaining Efficiency and Flexibility in the Simple Temporal Problem. In Chittaro, L.; Goodwin, S.; Hamilton, H.; and Montanari, A., eds., *Proceedings of the Third International Workshop on Temporal Representation and Reasoning (TIME-96)*. IEEE Computer Society Press: Los Alamitos, CA.

Coles, A.; and Coles, A. J. 2007. Marvin: A Heuristic Search Planner with Online Macro-Action Learning. *Journal of Artificial Intelligence Research*, 28: 119–156.

Coles, A. J.; and Coles., A. I. 2014. PDDL+ Planning with Events and Linear Processes. In *Proceedings of the 24th International Conference on Automated Planning and Scheduling (ICAPS)*.

Costa, V. S.; Rocha, R.; and Damas, L. 2012. The YAP Prolog system. *Theory and Practice of Logic Programming*, 12(1-2): 5–34.

Cox, M. T.; and Dannenhauer, D. 2016. Goal transformation and goal reasoning. In *Proceedings of the 4th Workshop on Goal Reasoning at IJCAI-2016*.

Dechter, R. 2003. *Constraint processing*. Elsevier Morgan Kaufmann. ISBN 978-1-55860-890-0.

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

Gerevini, A.; and Long, D. 2005. Plan Constraints and Preferences in {PDDL3}. Technical report, Department of Electronics for Automation, University of Brescia, Ital.

Gerevini, A. E.; Saetti, A.; and Vallati, M. 2009. An Automatically Configurable Portfolio-based Planner with Macro-actions: PbP. In *Proceedings of the 19th International Conference on International Conference on Automated Planning and Scheduling (ICAPS)*, ICAPS'09, 350–353. AAAI Press. ISBN 978-1-57735-406-2.

Getoor, L.; and Taskbar, B. 2007. *Introduction to Statistical Relational Learning*. The MIT Press. ISBN 0262072882.

Ghallab, M.; Howe, A.; Knoblock, C.; McDermott, D.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL - The Planning Domain Definition Language. Technical report, CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: Theory and Practice*. Morgan Kaufmann.

Gil, Y. 1992. *Acquiring domain knowledge for planning by experimentation*. Ph.D. thesis, CMU, Pittsburgh, PA, USA.

Gregory, P.; Long, D.; Fox, M.; and Beck, J. C. 2012. Planning Modulo Theories: Extending the Planning Paradigm. In *Proceedings of the 22nd International Conference on Automated Planning and Scheduling (ICAPS)*.

Helmert, M. 2006. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26(1): 191–246.

Hoffmann, J. 2001. FF: The Fast-Forward Planning System. *AI Magazine*, 22: 57–62.

Jiménez, S.; De La Rosa, T.; Fernández, S.; Fernández, F.; and Borrajo, D. 2012. A review of machine learning for automated planning. *The Knowledge Engineering Review*, 27(4): 433–467.

Mitchell, T. M. 1997. *Machine Learning*. New York, NY, USA: McGraw-Hill, Inc., 1 edition. ISBN 0070428077, 9780070428072.

Murphy, K. P. 2012. *Machine Learning: A Probabilistic Perspective*. The MIT Press. ISBN 0262018020.

Shen, W.-M.; and Simon, H. A. 1989. Rule creation and rule learning through environmental exploration. In *In Proceedings of the International Joint Conference on Artificial Intelligence*, 675–680.

Thayer, J. T.; Dionne, A. J.; and Ruml, W. 2011. Learning Inadmissible Heuristics During Search. In Bacchus, F.; Domshlak, C.; Edelkamp, S.; and Helmert, M., eds., *Proceedings of the 21st International Conference on Automated Planning and Scheduling (ICAPS)*. AAAI.

Tong, S. 2001. *Active Learning: Theory and Applications*. Ph.D. thesis, Stanford University.

Vattam, S.; Klenk, M.; Molineaux, M.; and Aha, D. W. 2013. Breadth of Approaches to Goal Reasoning : A Research Survey. *Goal Reasoning: Papers from the ACS Workshop*, 111.

Wang, X. 1994. Learning Planning Operators by Observation and Practice. In *International Conference on Artificial Intelligence Planning Systems*.