Goal-based Composition of Hybrid AI Systems

Uwe Köckemann[a,*]

[a]Örebro University
ORCID (Uwe Köckemann): https://orcid.org/0000-0001-7776-2116

Abstract. Integrated AI systems can often be composed of a series of black-box components. We propose a planning based approach that takes a set of components, initially available models/data, and set of goal models and automatically devices a plan that represents an integrated AI system. This allows us to automatically adjust to new components and changing requirements. Experimental evaluation shows how our approach performs over a large number of instances covering sets of over 100 components.

1 Introduction

Uses cases for AI often require the composition of multiple AI components (models, algorithms) into a hybrid system to achieve their goals. Systems such as AI Builder [15] allow to compose AI systems from individual components in the form of graphs. Each component may have multiple inputs and outputs and (partially) solves an AI (or related) problem when called. However, a comprehensive library of AI components would provide many choices for each problem and different sequences of steps may lead to the same outcome but in very different ways. For this reason, we would prefer to simply provide a system's initially available data, a set of available components, and describe the output of the system in order to allow an AI algorithm to take care of system composition for us.

Our approach transforms descriptions of available AI components into planning operators which allows us to use classical planning systems to create a system. to achieve this, we limit ourselves to systems described as *Directed Acyclic Graphs (DAGs)*. Experimental evaluation shows how our approach performs over a large number of instances covering sets of over 100 components.

Our approach is implemented in the *AI Domain Definition Language (AIDDL) [9]* [1] framework for integrative AI which allows to model the composition problem, planning problem, and our experiments in a common language and easily define translations where needed.

The remainder of this paper is organized as follows. Section 6 discusses related work. Section 2 gives a brief background on automated planning. Section 3 introduced the problem of composing hybrid AI systems. Section 4 shows how the composition problem can be turned into a planning problem. Section 5 presents preliminary results on randomly generated sets of components. Finally, Section 8 concludes the paper and discusses future work.

[*] Corresponding Author. Email: uwe.kockemann@oru.se.
[1] The AI Domain Definition Language (aiddl.org)

2 Background

We use a state-variable-based notion of automated planning [2]. States and goals are sets of variable-value pairs $x \leftarrow v$ where each state variable x is assigned at most one value v. Operators define how states can be changed. Each operator $o = (name, P, E)$ has a name, a set of preconditions P and effects E. An operator is applicable to a state s iff $P \subseteq s$. The state transition function γ applies an operator o to a state s and yields the following state by replacing all variable assignments that appear in the effects of o

$$\gamma(o, s) = (s \setminus \{x \leftarrow v | x \leftarrow v' \in E_o\}) \cup E$$

A plan π is a sequence of actions $\pi = a_1, \dots a_n$ that can be applied to a state s_0 generating a sequence of states

$$s_i = \gamma(a_i, s_{i-1}),$$

assuming $P_i \subseteq s_{i-1}$. A planning problem (s_o, g, O) is composed of an initial state s_o, goal g, and a set of operators O. A plan π is called a solution if $g \subseteq s_n$ and $\forall_i : a_i \in O$. In practice, operator use parameters which allows to specify them as templates for many possible instances.

Planning problems can be solved via state-space search [4] where each node is a state s and each edge is an action a leading from a state where the action is applicable to $\gamma(a, s)$. State of the art planning approaches rely on heuristics to inform which nodes are chosen next and are often solved with greedy best-first search.

3 AI Orchestration Problem

In this section we define AI system composition as its own problem where we have to select and apply a set of components to transform available inputs into desired outputs. This formulation provides a common baseline that allows studying different approaches to solving the problem (different planning approaches or alternatives based on constraint satisfaction [5] or Boolean satisfiability).

Let C be a set of components $c = (name, I, O, cond)$, where $name$ is a name, I and O are inputs and outputs respectively, and $cond$ is a set of constraints allowing to connect properties of inputs and outputs. Each element in I and O is a tuple (x, P) where x is a variable identifying the associated message, and P is a set or required properties for the inputs in I and asserted properties for the outputs in O. Components may be applied to different combinations of inputs and for each such combination, the outputs are considered different. Properties are used to indicate types (e.g., planning problem), and additional aspects (e.g., grounded, normalized). For the purpose of this paper, we assume that properties are symbolic. Figure 1 shows two example components. The first component is a grounder for planning problems. It takes an input $?I$ with the property *planning-problem* and produces an output $?O$ with the properties *planning-problem* and *ground*. The second component is a forward search planner which takes an input $?I$ with properties *planning-problem* and *ground* and produces an output $?O$ with the properties *plan* and *sequential*. Both components are expressed as AIDDL terms in form of tuples $(...)$ of key-value pairs $(k : v)$, which follows our definition of components.

Figure 1. Two example components for grounding and solving planning problems. $?I$ and $?O$ are input and output variables mapped to sets of properties. The examples are written in AIDDL. The colons (:) indicate key-value pairs.

```
(
  name:planning-problem-grounder
  inputs:{
    (?I {planning-problem})
  }
  outputs:{
    (?O {planning-problem ground})
  }
)

(
  name:forward-search-planner
  inputs:{
    (?I {planning-problem ground})
  }
  outputs:[
    (?O {plan sequential})
  ]
)
```

To execute a component, a fitting source is supplied for all required inputs in I. We assume that all messages are stored in a common memory that can be read and written by components as needed. After execution, all output messages are written into this memory so they can be accessed by other components given their address.

The orchestration problem is a tuple (A, Ω, C), where $\mathrm{A} = (\alpha, \emptyset, O, \mathit{cond})$ is a component with no inputs providing initially available messages (here, cond is used to assert initial facts such as the source relations mentioned below), and $\Omega = (\omega, I, \emptyset, \mathit{cond})$ is a component with no outputs describing the desired (goal) data (here, cond is used as normal to express relations between the inputs and conditions on source relations). C is a set of available components as described above.

A solution $\Pi = C^*, a$ to the composition problem is a set of selected components C^* and an assignment $a : X \to \mathbb{N}$ containing a mapping from all variables that appear in C^* to memory addresses identified by an integer. For convenience, we write C^*_Π to indicate C^* where all variable appearances have been replaced by their slots according the mapping a.

To express conditions in components, we define two relations that allow us to track where data comes from. The input relation indicates if some data x has been used to produce data y. The source relation can be used to model sources in the initially available data and propagate it when components are applied.

We define the input relation

$$\mathit{InputOf}(x, y) \iff \exists_{(\_, I, O, \_) \in C^*_\Pi} (x, \_) \in I \wedge (y, \_) \in O$$

to indicate that x was used as in input to compute y. The input relation allows to add constraints between inputs and outputs (e.g., a solution to a planning problem with specific properties, a decision tree based on normalized data-set).

We also define the source relation as

$$\mathit{Source}(x, s) \Leftarrow \mathit{InputOf}(x, y) \wedge \mathit{Source}(y, s).$$

We assume that this rule and the initial condition cond_A define all source relations. The purpose of this is to allow us to track where data comes from in order to clearly define the purpose of the system in Ω. As an example, we may have multiple states and sets of planning operators in a system and the source relation can be used constrain which operators can be used together with which states.

An *execution strategy* for C^*_Π is an ordering of all its components that guarantees that required data is available when a component is called. Not every C^*_Π can be executed. To see this consider the two component system

$$\begin{array}{lll} (A, & \{(1, \emptyset)\}, & \{(2, \emptyset)\}) \\ (B, & \{(2, \emptyset)\}, & \{(1, \emptyset)\}) \end{array}$$

where the components A and B each have one input and one output without any conditions. A and B provide and require each others' input and output which leads to a deadlock.

More generally, we can create a directed data dependency graph (V, E) where the set of nodes is the set of memory addresses used by C^*_Π. The set of edges $E = \{(x, y) | \mathit{InputOf}((, x), y)\}$ contains one edge between input and output slots for each operator.

If (V, E) is acyclic, the corresponding solution can be executed by executing each component following the topological ordering of the slots. If (V, E) is cyclic, we consider that it can be executed if we can order all slots in such a way that each slot is available with the required properties when it is first used. For instance, if we add a provider for slot 1 to the previous example we get the system

$$\begin{array}{lll} (A, & \{(1, \emptyset)\}, & \{(2, \emptyset)\}) \\ (B, & \{(2, \emptyset)\}, & \{(1, \emptyset)\}) \\ (C, & \emptyset, & \{(1, \emptyset)\}) \end{array}$$

which can be executed in the order C, A, B.

Our use of planning detailed in the following section avoids cycles, but we will investigate cyclic extensions in the future because they are relevant for many types of systems.

Given a composition problem (A, Ω, C), we can filter out irrelevant components. To do this, we initialized the set of relevant components with the ones in Ω. Then we repeatedly go through C and add any component that can provide the inputs to a relevant component as relevant. We stop when no new relevant components are found or when all components are marked as relevant.

4 Orchestration as an AI Planning Problem

Our aim is to use the problem defined in the previous section to create a planning problem whose solution will create a system C^*_x and an execution strategy for the system.

Our initial state will be created based on A, our goal state based on Ω, and we create one operator for each component in C, as well as some extra operators for propagating source relationships. A plan will be a sequence of actions representing component executions that require the data provided by A and produce the data required by Ω.

The signature of the operator is composed of its name (same as component name) followed by one variable per input. We create a precondition $\mathit{Property}(?I, x) \leftarrow \top$ for each required property of each input to indicate that the input must satisfy all required properties. Conditions cond of the component are assumed compatible in terms or representation and thus added directly.

Effects are created to assign properties to each output. Output slots are uniquely determined by the given name of the component, index of the output and the list of chosen input slots[2]. So for a component named c with chosen inputs i_1, i_2, the second output would receive slot $o_2 = (c, 2, [i_1, i_2])$. This assumes that the same inputs will always generate the same outputs.

Figure 2 shows the operator created for the forward planning service from Figure 1.

Figure 2. Planning operator created for the forward search planning service (with a shortened name) shown in Figure 1. Note that the output slot is a composition of the service name and a list of used inputs.

```
(name:(planning ?I)
 preconditions:{
   (property ?I planning-problem):true
   (property ?I ground):true
 }
 effects:{
   (property (planning [?I]) plan):true
   (property (planning [?I]) sequential):true
   (input-of ?I (forward-search-planning [?I]))
 }
)
```

The goal includes one property for each required output. Goal slots are created and available in the initial state to allow linking them to outputs that achieve the desired properties. To achieve goals, we use two additional operators *commit-to-slot* and *add-property* that allow to add properties to goal slots.

Figure 3. Operator that allows to commit a single goal property to a free goal slot in order to achieve a goal.

```
(
  name:(commit-to-slot ?E_in ?Prop ?E_out)
  preconditions:{
      (is-free ?E_out):true
      (property ?E_in ?Prop):true
      (is-goal-property ?Prop):true
  }
  effects:{
      (is-free ?E_out):false
      (input-of ?E_in ?E_out):true
      (property ?E_out ?Prop):true
  }
)
```

The execution strategy for sequential plans is straight forward. For each action that represents a component call, we execute the corresponding component and store its outputs under the chosen

Figure 4. Operator that allows to add further goal properties to a free goal slot (only usable after *commit-to-slot*).

```
(
  name:(add-property ?E_in ?Prop ?E_out)
  preconditions:{
      (input-of ?E_in ?E_out):true
      (is-free ?E_out):false
      (property ?E_in ?Prop):true
      (is-goal-property ?Prop):true
  }
  effects:{
      (is-free ?E_out):false
      (property ?E_out ?Prop):true
  }
)
```

slot names. Operators not representing components can simply be skipped.

In many cases it might be possible to parallelize execution. This can be achieved for sequential plans through de-ordering, or by relying on planning approaches that produce parallel or partially-ordered plans.

5 Evaluation

5.1 Implementation

The approach is fully implemented in the *AI Domain Definition Language (AIDDL)* for integrated AI systems which allows us to write model components, planning problems, and to define experiments.

We compare two settings for a state-variable based forward heuristic A* search with the (non-admissible) *Fast Forward (FF)* [6]. We test $w = 0.5$ (considering *Path Length PL*) and heuristic value with equal weight) and $w = 1.0$ (*No Path Length (NPL)*). In practice, including path length may lead to less complex systems using fewer components. Note however, that due to the way in which we generate instances (see next section), plan length can actually not vary in our experiments since it is determined by the number of layers.

The AIDDL framework integrates an experiment runner that allows to specify configurations and changing parameters for random generation, as well as target measures and approaches to be compared. This makes it very easy to re-produce or extend our experiments. Every generated instance, solutions, and all measured data are stored. Figure 5 shows the experiment configuration. The base configuration *base-config* is used to generate instances. It includes the variable NL which is varied for each set of instances. The variables entry defines possible values for each such variable. If multiple variables are present, a set for each combination of values will be created. The entry *num-instances* defines how many instances are created for each set.

5.2 Random Instance Generation

The instance generator creates random components in layers that rely on the previous layer and produce data for the next layer. So a component in layer n relies on data provided by components in layer $n-1$ and provides data for components in layer $n+1$. We always use a goal that requires an output from the last layer. This allows us to control problem difficulty because the solution length depends on the number of layers.

[2] An alternative version that managed a pool of available slots performed worse during planning. The reason could be that the output slots were required as part of the operator signature which increased the size of the set of ground operators significantly.

To generate an instance, we specify the number of layers, how many properties and components exist on each layer, as well as the minimum and maximum inputs and outputs for each component. Overall we created 25 sets (different numbers of layers) with 20 instance each. In this paper we stick to components with single inputs and outputs. We can show that a component with n outputs can be replaced by n components with single outputs. In a similar way, components with three inputs can be transformed into two components with two inputs each (combining two inputs into an intermediate result). If we assume that components can have a memory that can store previous inputs, we can reduce the number of inputs down with the idea that the first call simply stores the result and the second call takes its input together with the stored result.

Figure 5. AIDDL specification of the experiment.

```
rng-seed:1234
base-config:[
   num-layers:?NL
   num-properties-per-layer:5
   min-components-per-layer:5
   max-components-per-layer:7
   min-inputs:1
   max-inputs:1
   min-outputs:1
   max-outputs:1
   num-goals:1
 ]
data-module:data
num-instances:20
instance-type:term
variables:[?NL:[1 2 3 4 5 6 7 8 9 10 11
                12 13 14 15 16 17 18 19
                20 21 22 23 24 25]]
variable-names:[NL]
measures:[time-solve time-init
          nodes-seen nodes-opened
          nodes-closed
          num-ground-operators
          num-components
          num-relevant-components
          plan-length]
```

5.3 *Results*

We measured the time to initialize and solve the search problem, number of search nodes opened, closed, and seen, the number of components, number of relevant components, the number of ground operators, and the plan length. Here we report on the planning time, number of components, and opened search nodes. The number of opened search nodes gives a platform independent indicator of the search complexity the other two node metrics (closed and seen) scale in a similar way. Plan length depends on the number of layers because we need one action per layer in each problem.

The number of components gives an indication of the difficulty of the problem, the planning times shows that problems are solved within reasonable time frame, and the nodes opened give a machine independent idea of the complexity of the search space.

In the following box plots, we indicate the median value, 25th and 75th percentile, as well as the minimum and maximum values.

Figure 6 shows how many (relevant) components were generated for each instance set. According to our instance generation, this is the main source of difficulty. We can see that the amount of relevant components on average is reduced compared to the total number of components, but the variability increases with the number of layers.

Figure 7 shows the planning time against the number of generated layers. Figure 8 shows the number of nodes added to the open list of the search. We can see that planning times stays below 100 seconds even for 25 layers where the average number of relevant components exceeds 100 which shows that the approach can scale to reasonably sized systems.

6 Related Work

Web Service Composition (WCS) [11] takes as input a set of services, a desired new service, and semantic knowledge in form of an ontology that decides whether a chain of service calls is allowed. As a result, WCS requires a complex reasoning step to verify if a composition is legal. Our approach avoids this by stating properties and allowing the propagation of sources directly as part of planning operators. While less expressive, a significant set of AI systems can be expressed already in this way. [7] suggest a heuristic that considers semantic knowledge about services represented in an ontology to improve the search for web-service compositions. GoalMorph [16] transforms goals in case not all goals can be reached in a web-service composition problem. The approach by [12] integrate description logic reasoning into a planning algorithm.

Four shortcomings of heuristic search for web-service composition have been highlighted by [14]. Out of these four, parallel control flow and creation of variables (here: slots) have been addressed in Sections 3 and 4 respectively. Issues relating to uncertainty about the initial state (here: $Alpha$) and goal (here: Ω) do not concern us, since our approach does not look into the data. This makes our notion of composition less general since we cannot branch depending on concrete outputs. On the other hand, we can use off-the-shelf planning algorithms to compose systems.

Configuration Planning (CP) [13, 10] considers planning problems that depend on the availability of information (information dependencies) and require certain information to be available (information goals). There are some similarities between CP and our approach in that information is treated as available or not but its content is not considered. However, in CP the type of information is sufficient whereas here we support more complex scenarios by supporting properties of service inputs and outputs. This works well for AI applications, where often types remain the same, but important properties change. On the other hand, configuration planning may benefit from temporal constraints (e.g., information availability during a time interval) which are not relevant to the presented model of AI system composition.

7 Ethical Considerations

Handcrafted AI systems may already include optimization functions that reinforce social injustice. This problem may be amplified or become harder to catch when such systems are automatically created.

We have to carefully consider under which circumstances we allow an AI system to be deployed automatically. If its decisions can negatively impact people, careful vetting by domain experts must be used as a safe-guard. Mixed-initiative and explainable planning approaches may provide useful in this regard. Automatically analyzing plans that represent AI systems for potentially dangerous biases

Figure 6. Number of generated and relevant components for varying number of layers.

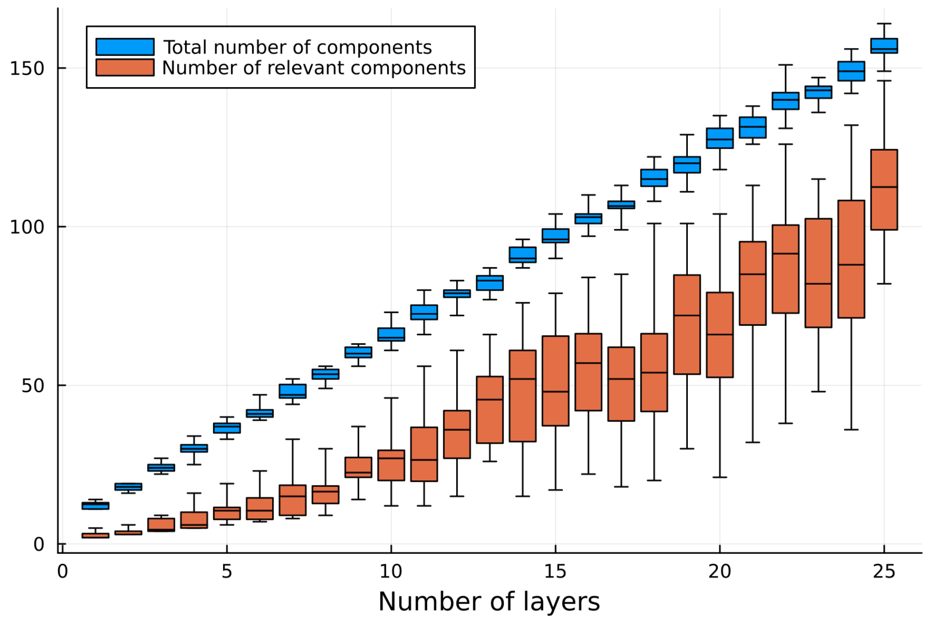


Figure 7. Planning times (logarithmic) vs number of layers.

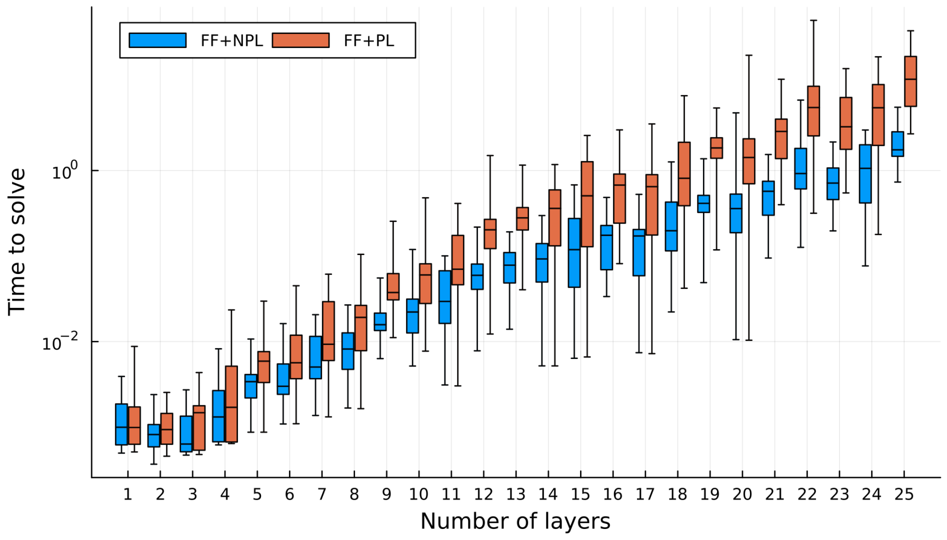
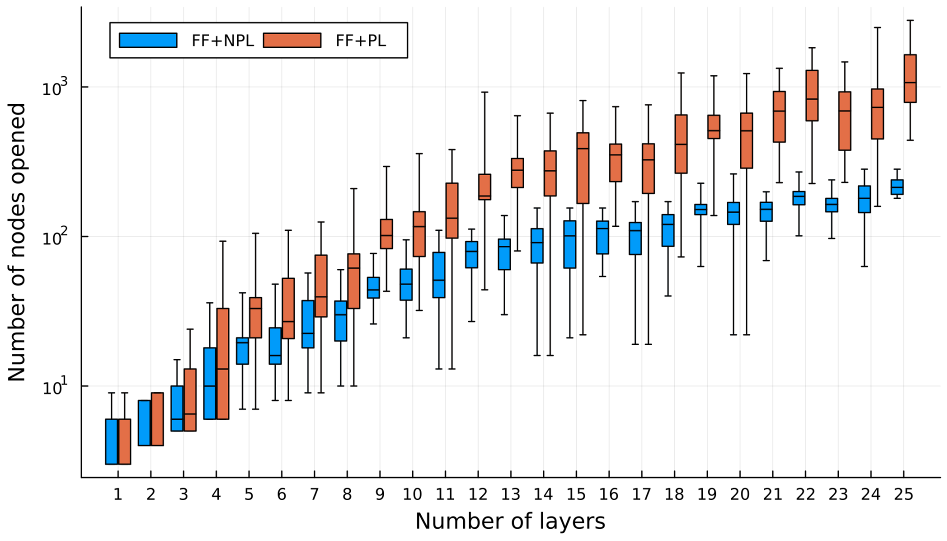


Figure 8. Number of nodes added to the open list of the search (logarithmic) vs number of layers.



based on the components that it uses would also make an interesting area for future research.

8 Conclusion & Future Work

We presented a planning-based approach for automatically building AI systems from user input and constraints. This work is still a prototype but early experiments show that it can compose reasonably sized systems. As pointed out earlier, there are various limitations in this approach due to the usage of classical planning (sequential plans, no cycles) and we will investigate execution strategies that could account for deadlock-free loops.

In many cases there are multiple options for fulfilling the same request. We plan to extend our representation to support preference-based planning [3]. In some cases, we may also want to automatically deploy a candidate system in order to evaluate it and find a good system considering multiple criteria (e.g., number of parameters, performance, computational resource requirements/costs). Hybrid planning approaches that include information about time and resources could be interesting for time critical systems or systems that may have to share computational resources such as GPUs. Finally, due to the ethical considerations pointed out above, we would like to investigate integration with mixed-initiative [1] and explainable planning [8] approaches where our approach is used together with a human operator rather than fully autonomously. Practically, we intend to deploy our approach on the AI Builder [3].

Acknowledgements

This work was supported by the AI4Europe project funded by the European Commission under Grant agreement Nr. 101070000. Many thanks to Jennifer Renoux for her help with processing of result data.

[3] https://ai-builder.aiod.eu

References

[1] M. Ai-Chang, J. Bresina, L. Charest, A. Chase, J. C. J. Hsu, A. Jonsson, B. Kanefsky, P. Morris, K. Rajan, J. Yglesias, B. G. Chafin, W. C. Dias, and P. F. Maldague. Mapgen: mixed-initiative planning and scheduling for the mars exploration rover mission. *IEEE Intelligent Systems*, 19(1): 8–12, Jan 2004. ISSN 1541-1672. doi: 10.1109/MIS.2004.1265878.

[2] C. Bäckström and B. Nebel. Complexity Results for {SAS+} Planning. *Computational Intelligence*, 11:625–655, 1995.

[3] J. A. Baier and S. A. McIlraith. Planning with Preferences. *AI Magazine*, 29(4):25–36, 2008. URL http://www.aaai.org/ojs/index.php/aimagazine/article/view/2204.

[4] B. Bonet and H. Geffner. Planning as heuristic search. *Artificial Intelligence*, 129(1):5–33, 2001. ISSN 0004-3702. doi: https://doi.org/10.1016/S0004-3702(01)00108-4. URL https://www.sciencedirect.com/science/article/pii/S0004370201001084.

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

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

[7] J. Hoffmann, I. Weber, J. Scicluna, T. Kaczmarek, and A. Ankolekar. Combining Scalability and Expressivity in the Automatic Composition of Semantic Web Services. In *Proceedings of the 2008 Eighth International Conference on Web Engineering*, ICWE '08, pages 98–107, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3261-5. doi: 10.1109/ICWE.2008.8. URL http://dx.doi.org/10.1109/ICWE.2008.8.

[8] S. Kambhampati. Synthesizing explainable behavior for human-ai collaboration. In *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, AAMAS '19, 2019.

[9] U. Köckemann. The ai domain definition language (aiddl) for integrated systems. In U. Schmid, F. Klügl, and D. Wolter, editors, *KI 2020: Advances in Artificial Intelligence*, pages 348–352, Cham, 2020. Springer International Publishing. ISBN 978-3-030-58285-2.

[10] U. Köckemann and L. Karlsson. Configuration planning with temporal constraints. In *Proceedings of the 31st AAAI Conference on Artificial Intelligence (AAAI)*, 2017. URL https://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14609.

[11] A. L. Lemos, F. Daniel, and B. Benatallah. Web service composition: A survey of techniques and tools. *ACM Comput. Surv.*, 48(3), dec 2015. ISSN 0360-0300. doi: 10.1145/2831270. URL https://doi.org/10.1145/2831270.

[12] Z. Liu, A. Ranganathan, and A. Riabov. A planning approach for message-oriented semantic web service composition. In *AAAI Conference on Artificial Intelligence*, 2007. URL https://api.semanticscholar.org/CorpusID:3734646.

[13] R. Lundh, L. Karlsson, and A. Saffiotti. Dynamic self-configuration of an ecology of robots. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3403–3409, 2007.

[14] H. Meyer and M. Weske. Automated service composition using heuristic search. In S. Dustdar, J. L. Fiadeiro, and A. P. Sheth, editors, *Business Process Management*, pages 81–96, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-38903-3.

[15] P. Schüller, J. P. Costeira, J. Crowley, J. Grosinger, F. Ingrand, U. Köckemann, A. Saffiotti, and M. Welss. Composing Complex and Hybrid AI Solutions. *ArXiv*, abs/2202.12566.

[16] M. Vukovic and P. Robinson. GoalMorph: Partial Goal Satisfaction for Flexible Service Composition. In *International Conference on Next Generation Web Services Practices (NWeSP'05)*, pages 149–154. IEEE, 2005. ISBN 0-7695-2452-4. doi: 10.1109/NWESP.2005.44. URL http://ieeexplore.ieee.org/document/1592421/.