Learning and Exploiting Decompositions in Automated Reasoning

1 Introduction

In this proposal, we outline our strategy to push the boundaries of planning and verification by pioneering innovative decomposition and compositional analysis techniques and algorithms. Our primary objective is to elevate the performance of decomposition methods and substantially enhance reasoning capabilities in these domains.

Traditional planning and verification techniques treat all system states uniformly, disregarding the internal structures and nuances of these states. While this approach offers flexibility in planning and enhances verification accuracy, it comes at the cost of potentially dealing with vast state spaces and extensive exploration paths. A burgeoning research direction in recent years is the exploration of internal symmetries, structures, and recurring patterns within systems, which can be harnessed to circumvent the exhaustive examination of all possible paths and states.

In particular, our focus is investigating how systems can be systematically decomposed to enable compositional analysis. This entails the ability to analyze a composed system while simultaneously establishing the properties of its components. We will develop and employ advanced inference rules that consider the semantics of the composition, allowing for a more efficient and effective approach to planning, verification, and analysis.

The core concept underlying our approach is the shift from solving a single, large, and intricate problem to addressing numerous smaller and more manageable problems. Each of these smaller problems is comparatively simpler to solve when contrasted with the larger overarching challenge. We will employ advanced learning techniques to break down complex systems into their constituent parts, considering various composition semantics, as illustrated in the two following examples. Additionally, we will employ advanced inference tools to reason about the properties of the composed system based on the insights gleaned from analyzing its components. Our primary goal is to develop robust and efficient methods for learning and effectively utilizing decomposition within the context of planning, synthesis, and model-checking challenges. In pursuit of these objectives, we will harness the power of Description Logic, Temporal Logic, and Formal Methods, which are mathematical and logical techniques for specifying and tackling the complexities of planning, synthesis, and model-checking tasks.

Through this research initiative, we intend to contribute to the broader field of artificial intelligence and computational problem-solving, ultimately leading to more effective and practical solutions for real-world challenges. Our dedication to developing these advanced methods signifies our commitment to pushing the boundaries of what is currently achievable in AI and search algorithms, with the ultimate goal of driving technological progress and innovation. Our study will focus on four main objectives:

1. **Compositional Analysis:** To enhance our ability to reason about the composed system, we will create heuristics and techniques that consider the interaction and dependencies among its components. These heuristics will facilitate more nuanced and insightful compositional analyses.

2. **Automatic Decomposition:** We aim to develop automated methods for effectively decomposing complex systems into their constituent elements. By automating this process, we seek to reduce the manual effort required and enable more efficient analysis and reasoning.

3. **Learning Domain-Specific Patterns:** Recognizing that different domains exhibit unique characteristics, our research will delve into the development of algorithms and tools that can identify and learn domain-specific patterns. These patterns may include symmetries, structures, or recurring behaviors, which can be leveraged to optimize planning and verification processes.

4. **Techniques for Exploiting Learned Patterns:** Once we've identified and learned these domain-specific patterns, we will explore innovative techniques for exploiting these patterns to our advantage. This could involve creating specialized algorithms, inference rules, or strategies that use the learned insights to streamline planning, verification, and synthesis challenges in a domain-specific context.

2 Two Motivating Examples

To demonstrate our research techniques, we present two representative research activities in detail. We chose to present one example in planning and one example in model-checking to demonstrate the breadth of our approach.

Example: Subtask decomposition by learning description logic formula

The greedy best-first search (GBFS) algorithm is a classic search algorithm that uses a heuristic function to determine the most promising path. Although simple, fast, and efficient, this algorithm is not always accurate and sometimes converges to local maxima. The paper [21] showed that the search space of GBFS with a given heuristic h induces a *bench transition system* (BTS) in which *benches* are connected via *progress stats* and *bench entry states*, which are all components of the search space that one may carefully use to improve planning techniques. Figures 1 and 2 illustrate the idea with an example from Heusner et al. on a toy domain. In Figure 1, a search space is presented with the heuristic values needed to determine which of the states are considered to be progress states (progress in the sense that, if such a state is encountered during the search, it might be best to continue from that point and not skip to another state). In Figure 2, a search space is presented with the heuristic values needed to split it into benches according to the progress states such that for each bench, we move forward to another bench via an "exit" progress state. Before our work [10], other methods could identify progress states only for a single task, and only *after* a solution for the task has been found. We surpassed this limitation and are able to learn progress states and use this knowledge *during* the search. Learning the entire BTS is a huge step toward decomposing planning tasks into subtasks. In this study, we plan to achieve the following two goals:

1. Learn a generalized representation of the BTS for a given domain and heuristic based on data from small instances;

2. Exploit the learned BST by performing a sequence of searches from a bench entry state to a progress state.

In Heusner et al. [21] and other studies, the BTS is not learned but deduced afterwards when the solution (the plan) has already been found. Our main goal is to use those benches, break down the search into intermediate subsearches, and ultimately achieve subplans such that, if combined sequentially, produce one plan. In [9], there is a similar approach; however, it uses a different technique that is limited to some specific domains.

To decompose planning problems into subtasks, the suggested research will use a technique based on decision trees to learn the BTS and represent them using description logic formulas. The learning process is performed for each domain from small examples (small instances), and then the learned BTS is used to solve large instances of the same domain.

A high-level overview of our approach is as follows: First, all benches for each task are generated, and then the description logic features are generated. For each bench, a formula is learned that describes its exit states, and bench A is merged into bench B if the goal formula of B is also suitable for A. Lastly, for each bench, a formula is learned that describes a state in the bench. To use this framework, bench walking, i.e., an intermediate search between the learned benches, is performed.

This line of research is an extension of our successful study presented at IJCAI 2022 [10], in which the main goal was to introduce a novel approach that learns a description logic formula characterizing all progress states in a classical planning domain. Using the learned formulas in a GBFS to break ties in favor of progress states often significantly reduces the search effort. Our previous work showed that learning progress states is feasible and efficient. The next step is to not only resolve cases of tie-breaking but to take the power of learning progress states to learn the complete BTS and improve the search itself. The immediate implication of the results attained and knowledge gained is the ability to decompose a planning problem into subproblems and solve it sequentially. Other work that is based on Sketches [9] showed that the decomposition technique is efficient and revolutionary. An additional secondary implication is related to the implementation; we aim to represent PDDL goals as description logic and vice versa, and this itself is a contribution.

Fields likely to benefit from our results are learning and automated planning, including but not limited to applications such as path and motion planning for autonomous robots, unmanned aerial vehicles, and autonomous driving. As to our latest publication [10], we guarantee that progress states can be learned and improve the expansion rate during search. In this proposal, we move to our next questions: Can a complete BTS be learned? Would it improve search? Would it be possible to perform a sequence of searches from a bench entry state to a progress state? Positive answers to these questions would mean a more efficient search, smaller expansion rate, and shorter run time. Of course, trade-offs should be considered and presented as part of the empirical analysis.

Example: Behavioral programming decomposition for efficient model checking

The behavioral programming (BP) paradigm is an approach for modeling and developing complex reactive systems, such as interactive games, robotic systems, or traffic control systems. BP allows the programmer to specify the system behavior as a collection of independent modules, called b-threads, that communicate and coordinate via events. Each model component, called a b-thread, can request, wait for, or block events, thus influencing the global event selection mechanism that determines the next system state. BP enables modular, incremental, and scenario-based development of reactive systems and easy debugging and testing. See [19] and the references therein for an overview of the approach.

To give another concrete example of the main methods we will use in the proposed research, we describe how BP decomposition and automated deduction techniques can enhance the efficiency of model checking of reactive systems. We illustrate this approach using an example scenario where multiple behavioral threads interact within

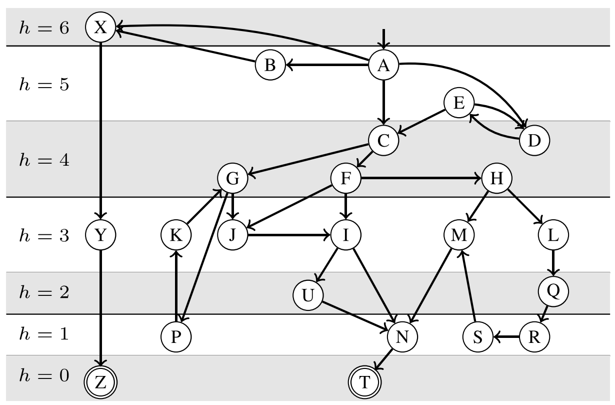

Figure 1: Search space topology from [21] for a toy domain, where the goal states are indicated by double lines.

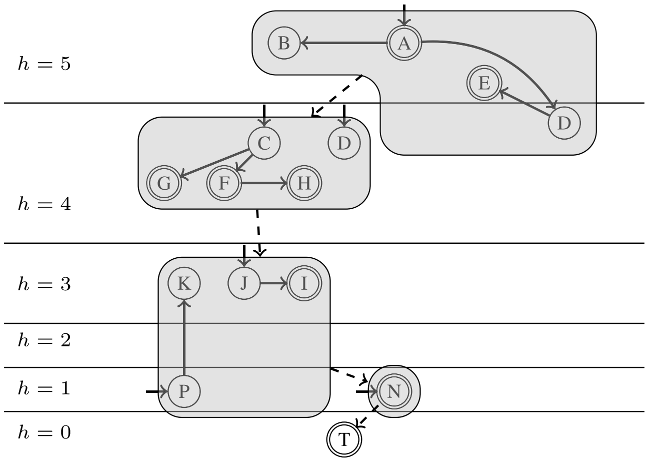

Figure 2: Search space topology from [21] for a toy domain, divided into benches connected via progress states. The progress states are indicated by double lines. Note that the progress states are actually the goal states for each bench.

a system. We aim to show how this method allows for significant computational savings, reducing the complexity from $o(p \cdot q)$ to $O(p + q)$ through BP and automatic logical reasoning.

Consider a reactive system composed of three key b-threads, each contributing to the system's overall behavior. First, b-thread A continuously requests an event, indicating a persistent need for a specific action. In addition, b-thread B blocks the event at times not divisible by the integer p, introducing a periodic and conditional interruption to the event's availability. Similarly, b-thread C plays its role by blocking the event at times not divisible by the integer q. These three b-threads function concurrently, intertwining their actions through BP interleaving. This concurrent operation forms the joint run of the b-threads, governing how they interact within the system. Scrutinizing the behaviors of these individual b-threads and their interactions enables advanced analysis techniques to verify the correctness and desired properties of the system efficiently.

Traditional model checking necessitates exploring the entire product space of these b-threads, which results in a time complexity of $o(p \cdot q)$. The key innovation in our approach is the application of behavioral decomposition and automatic mathematical deduction to mitigate this complexity. By examining each behavioral b-thread individually, we can significantly alleviate the computational load. An automatic analyzer can independently scrutinize the states of b-thread B and mechanically deduce that the event cannot be triggered at times that are not divisible by p. Similarly, it can separately analyze the state space of b-thread C and infer that the event cannot be triggered at times that are not divisible by q. Utilizing straightforward automatic reasoning with tools such as the Z3 theorem prover [5], it can be determined that the event can only occur at times divisible by $p \times q$ (assuming that both are prime numbers).

By leveraging behavioral decomposition and number theory, we reduce the overall complexity of model checking from $p(p \cdot q)$ to $O(p + q)$. This approach efficiently verifies reactive systems without exploring the entire product space. In previous work [17], we studied the possibility of achieving a succinct decomposition of systems. This research proved to be a significant step forward in understanding how to efficiently manage and manipulate complex reactive systems using BP. In the proposed research we will research how to combine it with other approaches to lighten verification such as GR1 [32], Spectra [1, 27] and approaches for hierarchical analysis [7].

Building on this successful work, our proposed research aims to develop compositional analysis techniques and automatic decomposition approaches. The goal is to leverage the succinctness of BP into the efficiency of different analysis tasks. By doing so, we hope to enhance the capabilities of BP, making it an even more powerful tool for managing the complexity inherent in reactive systems.

This research will not only build upon our previous work but also open new avenues for exploration in the field of BP. We believe our approach will contribute significantly to the ongoing efforts to improve reactive system development and analysis efficiency and effectiveness. This illustrative example demonstrates the potential of BP and decomposition techniques in the context of model checking. By focusing on individual behavioral threads and using number theory, we can achieve substantial computational savings, ultimately leading to more efficient and effective solutions for real-world challenges in automated reasoning and verification.

In the proposed research, as elaborated in subsequent sections of this proposal, we will explore applying various theories for compositional deduction, develop idioms for BP, and demonstrate real systems that benefit from this approach.

3 Background

3.1 Classical Planning

Throughout this proposal, we work with planning domains and tasks defined in the *Planning Domain Description Language* (PDDL) [29]. A *domain* is a tuple $\mathcal{D} = \langle \mathcal{P}, \mathcal{A}, \mathcal{C} \rangle$, where $\mathcal{P}$ is a set of predicate symbols (along with an arity); $\mathcal{A}$ is a set of action schemata and $\mathcal{C}$ are constants.

A *task* Π of domain $\mathcal{D}$ is a tuple $\langle \mathcal{O}, \mathcal{P}, A, s_\mathrm{I}, \delta \rangle$, where $\mathcal{O}$ is a set of objects and $\mathcal{P}$ is a set of first-order predicates. A *fact* refers to a predicate $p \in \mathcal{P}$ with arity k grounded to $p(o_1, o_2, \ldots, o_k)$ with $o_i \in \mathcal{O}$. Let F be the set of all facts. Then, any $s \subseteq F$ is called a state, and the set of all states $S(\Pi)$ is called a *state space*. Moreover, $s_\mathrm{I} \in S(\Pi)$ is the *initial state* and δ is the *goal condition*, a first-order logical formula over $\mathcal{P}$, $\mathcal{C}$ and $\mathcal{O}$. All states $s \supseteq \delta$ are *goal states*, and the set of all goal states is denoted $S_G(\Pi)$. A is a set of *action schemas* that can be grounded using $\mathcal{O}$. We call grounded action schemas *actions*. An action a is a tuple $\langle \mathit{pre}, \mathit{add}, \mathit{del} \rangle$ with $\mathit{pre}, \mathit{add}, \mathit{del} \subseteq F$ and is associated with a cost $\mathit{cost}(a) \in \mathbb{R}_0^+$. Action a is applicable in state s if $\mathit{pre} \subseteq s$. Applying a in s, written as $s[\![a]\!]$, leads to the successor state $(s \setminus \mathit{del}) \cup \mathit{add}$. An action sequence $\pi = \langle a_1, a_2, \ldots, a_n \rangle$ is applicable in state s if every action a_i is applicable in the state $s[\![a_1]\!][\![a_2]\!][\![\ldots]\!][\![a_{i-1}]\!]$. The cost of an action sequence is the summed-up cost of its actions. A state s' is *reachable* from s if there is an applicable action sequence starting in s and ending in s'. The *reachable state space* $S_R \subseteq S$ is the set of all states reachable from s_I. An applicable action sequence starting in state s and ending in a goal state is called an *s-plan*. The objective in classical planning is to find an s_I-plan, i.e., a *plan* for the given task.

3.2 Behavioral Programming

Behavioral programming (BP) is a paradigm that facilitates the modeling and development of complex reactive systems, such as interactive games, robotic systems, or traffic control systems. BP enables the programmer to define the system behavior as a collection of independent modules, known as b-threads, which communicate and coordinate via events. Each b-thread can perform three basic operations on events: request, watch, and block. These operations allow b-threads to express their preferences and constraints on the system behavior, and to cooperate or compete with each other to achieve the desired system goals.

The core concepts of BP are formally defined as follows:

- **b-thread**: A b-thread is a Labeled Transition System (LTS) with three state labeling functions. An LTS is a triple $TS = (Q, Lab, \rightarrow)$, where Q is the finite set of states, Lab is the finite set of labels (events), and $\rightarrow \subseteq Q \times Lab \times Q$ is the transition relation. In the context of a b-thread, each state in Q is labeled with three functions: R (requested events), W (waited-for events), and B (blocked events). The transition relation $\rightarrow$ defines how the b-thread transitions from one state to another in response to events.

- **b-program**: A b-program is a collection of b-threads. If we denote the ith b-thread as $BT_i = (Q_i, Lab_i, \rightarrow_i, R_i, W_i, B_i)$, then a b-program BP can be represented as a set of b-threads, i.e., $BP = \{BT_1, BT_2, ..., BT_n\}$.

- **Composition semantics**: The composition semantics of a b-program are defined by the set of all possible interleavings of the events triggered by its b-threads. Each run is a sequence of events resulting from the state evolution as defined by the transition relations of the b-threads. The composition semantics thus define the behavior of the b-program as a whole. The event triggered in each step must be requested and not blocked, and each b-thread that waited for the selected event advances accordingly while the other b-threads remain in their states without moving. Formally, the semantics are defined by the sequences of events consistent with the following composed LTS: $(s_1, \ldots, s_n) \xrightarrow{e} (s'_1, \ldots, s'_n)$ if and only if $e \in R_1(s_1) \cup \cdots \cup R_n(s_n)$ and $e \notin B_1(s_1) \cup \cdots \cup B_n(s_n)$ and $s'_i = s_i$ if $e \notin W_i(s_i)$ and $s_i \xrightarrow{e}_i s'_i$ if $e \in W_i(s_i)$.

To put it simply, a b-thread can be thought of as a single player in a team, each following its own set of instructions or rules. A b-program is the entire team, where each player (b-thread) collaborates to achieve a common goal. The composition semantics serve as the rulebook that guides how the team plays together. It determines when each player can move and how their actions influence the game. This analogy can aid in understanding the intricate concepts of BP.

3.3 Description Logic

Description logic (DL) is a family of knowledge representation formalisms [2] that use the notions of *concepts*, which are classes of objects that share some property, and *roles*, which are the relations between these objects. Interpreting the concepts and roles for a planning state yields a *denotation*, i.e., a set of objects $O \subseteq \mathcal{O}$ for a concept, and a set of object pairs $\{\langle o_1, o_2\rangle, \langle o_3, o_4\rangle, \ldots\} \subseteq \mathcal{O} \times \mathcal{O}$ for a role.

Concepts and roles are recursively defined and interpreted for a state $s \in S$. At its base are the *universal concept* $\top$ and the *bottom concept* $\bot$ with semantics $\top(s) = \mathcal{O}$ and $\bot(s) = \emptyset$, as well as *atomic* concepts and roles. A atomic concept $C_{p,i}$ for a k-ary predicate $p \in \mathcal{P}$ and its i-th argument is interpreted in s as $C_{p,i}(s) = \{o_i \mid \exists o_1, \dots, o_k \text{ s.t. } p(o_1, \dots, o_k) \in s\}$.

Accordingly, an atomic role $R_{p,i,j}$ for a k-ary predicate $p \in \mathcal{P}$ and its i-th and j-th arguments is interpreted as $R_{p,i,j}(s) = \{\langle o_i, o_j\rangle \mid \exists o_1, \dots, o_k \text{ s.t. } p(o_1, \dots, o_k) \in s\}$ in s. Let X and X' be two concepts (respectively, two roles). They can be combined to form new concepts and roles via grammar rules. Examples are negation, union, and intersection, which are interpreted in a state s as $(\neg X)(s) = \mathcal{O} \setminus X(s) \quad \text{resp.} \quad (\neg X)(s) = \mathcal{O} \times \mathcal{O} \setminus X(s), (X \sqcup X')(s) = X(s) \cup X'(s), \text{ and} (X \sqcap X')(s) = X(s) \cap X'(s)$, respectively.

We use the same grammar as [13]. For details, we refer to their extended paper [14].

3.4 Decision Trees

A binary decision tree is a machine learning model with a binary tree structure [3]. Let $\mathcal{C}$ be a set of classes and let F be a list of features. A decision tree assigns a class $c \in \mathcal{C}$ to a vector $v \in \mathbb{R}^F$. Each internal tree node n_I is associated with a feature $f(n_I) \in \{1, \dots, F\}$ and threshold $\tau(n_I) \in \mathbb{R}$. Each leaf node n_L is associated with a class $c(n_L) \in \mathcal{C}$. To assign a class to an input vector v, the decision tree is traversed from the root node to a leaf node. At every internal node n_I, if $v[f(n_I)] \leq \tau(n_I)$, then the traversal continues to the first child node, otherwise it continues to the second one. When a leaf node n_L is reached, the input is labeled as $c(n_L)$.

Decision trees are greedily constructed given some training data $\langle D, L\rangle$ with feature matrix $D \in \mathbb{R}^{M \times F}$ and the label vector $L \in \mathcal{C}^M$, where M is the number of training samples. Each node n is associated with a non-exclusive submatrix $D_n \in \mathbb{R}^{M' \times F}$ and $L_n \in \mathbb{R}^{M'}$. The root node is associated with the whole training data D and L, and is initially a leaf node. Leaf node n_L is associated with the most frequent class in L_{n_L}. During training, the algorithm chooses a leaf node n_L and searches through combinations of features f' and thresholds τ', which are used to group data points $\langle D_{n_L}[i], L_{n_L}[i]\rangle$ for $i \in \{1, ..., M'\}$ into two sets using test $D_{n_L}[i][f'] \leq \tau'$. The quality of the groups is evaluated using a metric (e.g., the Gini impurity [3]). The leaf is associated with a combination of the best split ($f(n_L) = f'$ and $\tau(n_L) = \tau'$), and two child leaves are added to it, one per data set split. This transforms n_L into an internal node.

The algorithm continues until all leaves contain only labels from the same class or a maximum tree depth is reached.

4 Related Work

4.1 State Space Topology/Progress States

Based on Heusner et al.'s paper [22], let Π be a planning problem with a state s. A heuristic $h : S \to \mathbb{R}_0^+ \cup \{\infty\}$ estimates the cost of an optimal s-plan. Let P be the set of all acyclic s-plans. The *high-water mark* of s is the largest heuristic value that needs to be considered to reach a goal state from s. In [22], Heusner et al. defined a state s as *progress state* iff its high-water mark is higher than the high-water mark of at least one of its successor states. Counterintuitively, this definition excludes goal states for goal-aware heuristics.

Let $\mathcal{X}()$ denote a state space. A bench b is a set of states $s \in \mathcal{X}()$. Let $\mathcal{B}$ denote the set of all benches of $\mathcal{X}()$. Intuitively, we would expect $\mathcal{B}$ to be a partitioning of $\mathcal{X}(,)$ but this is not the case. By the original definition, states can be in multiple benches. For a bench $b \in \mathcal{B}$, *states*(b) denotes the states of b; *entry*(b) denotes the entry states of b; and *exit*(b) denotes the exit states of b. The level of a bench is denoted by *level*(b). In [22] the authors manually identify progress states and benches, in this proposal, our goal is to use the topology from [22] to decompose the state space and learn the different benches before execution.

4.2 Learning and Exploiting Progress States in GBFS

Theoretical properties of optimal state-space search algorithms like A* or IDA* have been extensively studied and are comparatively well understood [28, 31, 6, 25, 20, 24]. A corresponding theory for suboptimal search algorithms such as GBFS [8] has received growing attention only in the last few years [35, 36, 37, 21, 22].

The main insight of [21] is that every run of a GBFS can be partitioned into different episodes defined by so-called *high-water mark benches*, and the *state-space topology* can be partitioned in the same way. All states s on a bench share the same *high-water mark* value, *progress states* are states that must be expanded to reach the next high-water mark bench. Exploiting knowledge of high-water mark benches or progress states during search gives rise to many applications. The only known algorithm that computes high-water mark benches does so *a posteriori*, i.e., it computes the benches of a problem after a plan has been found [22]. At this point, the high-water mark information is not needed.

In [10], the authors proposed the following pipeline: For a given domain and heuristic, fully expand the reachable state spaces of several small tasks and annotate all states with their heuristic value. Using the heuristic values, determine whether each state is a progress state. Next, compute a set of description logic features and evaluate each on a subset of states. Then, adopt a decision tree [3] learning algorithm to learn simple formulas over the description logic features in disjunctive normal form (DNF), which predicts whether a state is a progress state. Finally, use the formulas to break ties in a greedy best-first search, demonstrating a use case for the trained progress state classifier.

Their method is evaluated using the h^{+} and h^{FF} heuristics [23], and they showed that the approach successfully learns useful formulas for identifying progress states. There is some trade-off between the quality of the formulas and the time required to evaluate them. However, they showed that exploiting progress states is beneficial: it significantly reduces the number of expansions required to find a plan.

4.3 Succinctness of Behavioral Programs

In our recent paper, “On the Succinctness of Idioms for Concurrent Programming” [17], we conducted an in-depth analysis of the efficiency of various concurrent programming idioms, particularly emphasizing their descriptive succinctness. Our study centered on three fundamental concurrent programming idioms: event requesting, blocking, and waiting. We found that a programming model that integrates all three idioms is exponentially more succinct than non-parallel automata. Furthermore, its succinctness complements classical nondeterministic and “and” automata.

This paper is relevant to this research proposal, which aims to use succinct decomposition methods to enhance analysis techniques such as model-checking and planning. Our findings offer a rigorous framework for evaluating the complexity of specifying, developing, and maintaining intricate concurrent software, aligning seamlessly with the proposal's objectives.

Our paper's exploration of the descriptive succinctness of automata and its implications for software reliability, maintainability, reusability, simplicity, and software analysis and verification could offer valuable insights for this research. Each idiom's unique succinctness advantages, which are not overshadowed by their counterparts, could be potentially harnessed in your proposed decomposition methods to boost the efficiency of model-checking and planning.

Our paper lays a solid foundation and offers valuable insights into the role of descriptive succinctness in concurrent programming. These insights could prove instrumental in our research on enhancing analysis through succinct decomposition methods.

5 Road map

We have structured our research into four distinct work packages, each with a specific focus. Work Package 1 is dedicated to breaking down reasoning problems into their fundamental components, leveraging generalized subtask systems. Simultaneously, Work Package 2 will operate in parallel, applying the insights gleaned from the first package to enhance heuristic search through the mastery of subtask systems by learning how to perform the decomposition automatically. In the realm of Work Package 3, our objective is to devise methodologies that autonomously construct models from data, employing process mining and automata learning techniques. Lastly, Work Package 4 is dedicated to the development of algorithms and tools for the compositional analysis of complex reactive systems. This involves scrutinizing individual components and deducing properties of the composed system. To execute these tasks, we will engage two or three Ph.D. and M.Sc. students for each task who will be recruited and mentored by us. The allocation of tasks will be tailored to the individual preferences of the students, ensuring an effective and collaborative approach to our research endeavors. The details of the work packages follow.

Work Package 1: Subtask Systems Decomposition in Planning Problems for a Specific Domain

This proposal's main idea is to decompose reasoning problems into its components. To do so, we need to formally define subtasks, generalized subtasks, and a system of subtasks. In the described case, instantiating a **generalized** subtask with a given state of a given instance yields a subtask, which is a PDDL task. In this work pack, we need to implement and formally define the following abstractions towards the learning phase together with heuristic and search analysis.

Generalized Subtask. For a planning domain $\mathcal{D} = \langle \mathcal{P}, \mathcal{A}, \mathcal{C} \rangle$, a generalized subtask encapsulates specific logical formulas $\phi_{membership}, \phi_{goal}$, for membership and goals within the planning domain. We denote it as $\mathcal{G}$.

Subtask. This describes a specific subtask for a state s within a planning task using a pre-defined generalized subtask for a given domain. We denote it as $\Pi_{\mathcal{G}}(s)$.

Generalized Subtask System. A set of generalized subtasks $\mathcal{S}$ for domain $\mathcal{D}$ is a *subtask system* for a set of tasks $\mathcal{T} = \{\Pi_1, \ldots, \Pi_n\}$ of $\mathcal{D}$ if for each $\Pi \in \mathcal{T}$ specific conditions holds.

The first step of our "bench walker" is to identify the initial state and bench to use for the next subsearch. In the simplest case, we have the current initial state s'_I and identify exactly one bench b that has s'_I as an inner state, and that we have not traversed in a circle. Then, we execute a subsearch for the pair $\langle s'_\mathrm{I}, b\rangle$.

In the second step, we set up and execute the subsearch. We first update the initial state of the task Π to s'_I. Then, we construct the next subgoal from the outer goal formula of the bench.

For each bench b, we calculate a formula ϕ_b, which identifies all the states of the bench. We exclude the exit states here, because they are also part of another bench as entry state and the correct subgoal is determined by the other bench. ϕ_b evaluates to true on a state $s \in \mathcal{X}()$ iff $s \in \mathit{states}(b) \setminus \mathit{exit}(b)$. To learn ϕ_b, we take all states $s \in \mathcal{X}()$ and label them as true iff $s \in b$. Now, we can use any approach – e.g., our description logic and decision tree technique – to learn the formula. Unfortunately, the learned formula can contain errors.

In addition, we associate each bench b with a formula δ_b that evaluates to true on all states $s \in \mathit{exit}(b)$ and to false on all states $s \in \mathit{states}(b) \setminus \mathit{exit}(b)$. To learn δ_b we use only the states $s \in \mathit{states}(b)$ and label the results as true iff $s \in \mathit{exit}(b)$.

Work Package 2: Learning Subtask Systems

This work pack is also related to the first example and illuminates its learning aspects. Heusner et al. [21] introduced high-water mark benches, which enable a run of GBFSs to be decomposed into several subsearches. So far, this decomposition has not been applied in practical tasks, as high-water mark benches can only be generated *a posteriori*, i.e., after a planning task has been solved. Here, we exploit the idea by learning a generalized subtask from benches that are generated on a set of tasks $\{\Pi_1, \ldots, \Pi_n\}$ from domain $\mathcal{D} = \langle \mathcal{P}, \mathcal{A}, \mathcal{C}\rangle$ that can be solved. We hope to show experimentally that the learned subtask systems generalize to large instances of the same domain in the empirical evaluation of the research. For now, we formally define such systems. Figure 3 is an example of a generalization over small instances; we plan to add more details as the research progresses. In the following, we describe the learning steps in detail.

Our approach consists of the following steps:

1. **Generate Benches:** generate all benches for each task;
2. **Generate Features:** generate the description logic features; that allow to (perfectly) describe all states on all benches and all exit states of all benches.
3. **Learn Goals:** for each bench, learn a formula that describes its exit states; separate exit states from all other states reachable from that bench
4. **Merge Benches:** merge bench A into bench B if the goal formula of B is also perfect for A;
5. **Learn Memberships:** for each bench, learn a formula that describes the states in the bench; separate member states from all other states

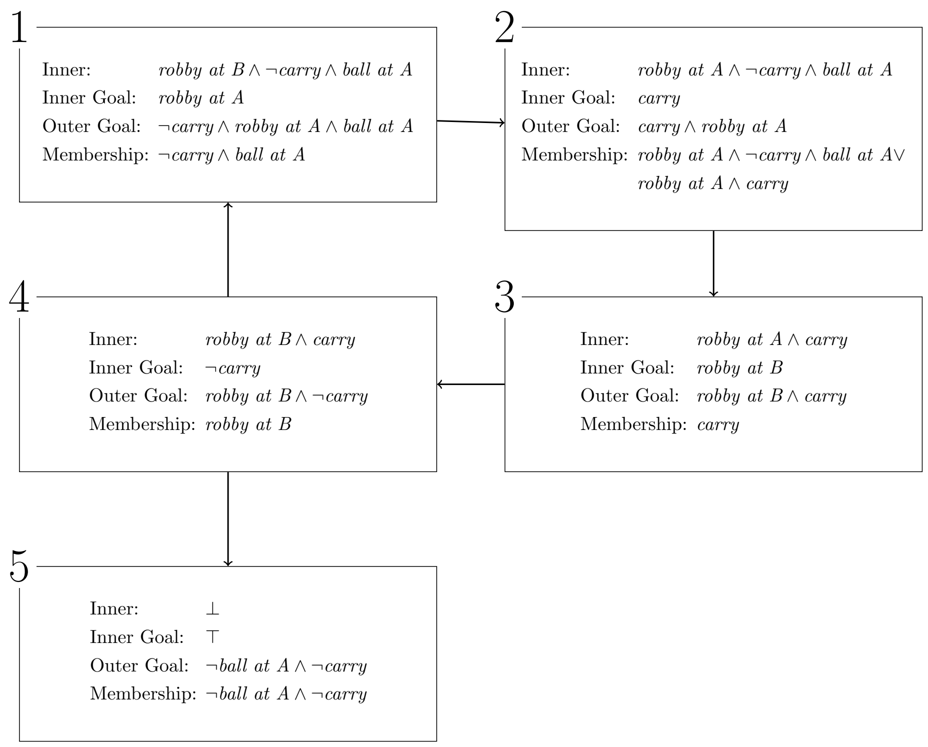
1
Inner: $robby\ at\ B \wedge \neg carry \wedge ball\ at\ A$
Inner Goal: $robby\ at\ A$
Outer Goal: $\neg carry \wedge robby\ at\ A \wedge ball\ at\ A$
Membership: $\neg carry \wedge ball\ at\ A$
2
Inner: $robby\ at\ A \wedge \neg carry \wedge ball\ at\ A$
Inner Goal: $carry$
Outer Goal: $carry \wedge robby\ at\ A$
Membership: $robby\ at\ A \wedge \neg carry \wedge ball\ at\ A \vee robby\ at\ A \wedge carry$
3
Inner: $robby\ at\ A \wedge carry$
Inner Goal: $robby\ at\ B$
Outer Goal: $robby\ at\ B \wedge carry$
Membership: $carry$
4
Inner: $robby\ at\ B \wedge carry$
Inner Goal: $\neg carry$
Outer Goal: $robby\ at\ B \wedge \neg carry$
Membership: $robby\ at\ B$
5
Inner: $\perp$
Inner Goal: $\top$
Outer Goal: $\neg ball\ at\ A \wedge \neg carry$
Membership: $\neg ball\ at\ A \wedge \neg carry$

Figure 3: GBTS for a toy domain, a gripper with one arm that needs to move balls between rooms using the h^* heuristic. In each box, a formula representing the relevant states is presented.

6. **Iterate:** if any learned formula is imperfect, return to description logic feature generation (step 2) with a higher complexity limit and repeat.

Generating Benches: For each input task Π_i, we label the states of a planning instance with respect to a heuristic h as progress or non-progress states. To do this, we first expand the reachable state space. For each state s, we track its predecessors $pred(s)$, successors $succ(s)$, and heuristic estimate $h(s)$. Note that this is done on small examples, where expending the search space is feasible.

In a second iteration, we compute the high-water marks (hwm). Initially, we set $hwm(g) = h(g)$ for all goal states g and regard $hwm(s)$ as undefined for all other states s. We maintain an open list of states ordered by high-water mark values that initially contain all goal states. Upon retrieving a state s from the open list, we insert all its predecessors $p \in pred(s)$ with undefined high-water marks into the open list with a high-water mark value of $hwm(p) = \max(h(p), hwm(s))$. The algorithm guarantees that a state is only inserted once into the open list, namely after its successor with the lowest high-water mark value has been retrieved from the open list. When this process terminates, only unsolvable states have an undefined high-water mark, which we treat as ∞ from here on.

Generating Features: In the second step, we generate the features used in our formulas to describe the membership and goals of the subtasks. The feature generation is an iterative process and is parameterized with a complexity limit ℓ. In iteration j, only features of complexity j are generated. In the first iteration, we only consider atomic concepts and roles over the predicates $\mathcal{P}$. In every succeeding iteration j, we additionally consider all features obtained by combining features from iteration $j-1$ with a rule from Section 3.3 if their combined complexity is equal to j. We prune any new feature f if its denotation is the same on all states $s \in S$ as the denotation of a previously considered feature. We stop the feature generation once we reach the complexity limit. Let $\mathcal{F}_\ell$ denote the set of considered features.

Learning Goals: For each bench $v \in V$, we compute a formula ϕ_{goal} that describes all exit states of the bench. Those exit states become subgoals of subtask. We generate the training samples by enumerating all pairs of states $\langle s, r\rangle$ such that $s \in v$ and r is reachable from s without traversing over an exit state. We label a pair $\langle s, r\rangle$ positively if r is an exit state and negatively otherwise. Furthermore, for each feature $f \in \mathcal{F}_\ell$ we compute whether its denotations on s and r as well as the set differences between those denotations are empty, i.e., $|f(s)| > 0, |f(r)| > 0, |f(s) \setminus f(r)| > 0, |f(r) \setminus f(s)| > 0$. As [10], we train a decision tree on those samples, extract a formula in disjunctive normal form from the tree, and simplify it using SymPy [30].

There is no guarantee that the generated features are sufficient to perfectly separate exit states from non-exit states because we might need a feature with a complexity higher than the complexity limit ℓ. In this case, we have to increase ℓ and start again.

Merging Benches: Let $v, v' \in V$ be two benches with goal formulas $\phi_{goal}(v)$ and $\phi_{goal}(v')$, respectively.

If these two benches represent the same subtask, then we replace $\phi_{goal}(v)$ with $\phi_{goal}(v')$ iff $\phi_{goal}(v')$ is more general than $\phi_{goal}(v)$ (same for $\phi_{avoid}(v)$).

Learning Membership: In this step, we learn formulas for $\phi_{membership}(v)$ for each $v \in V'$.

Theorem 1. *What we learn (under some assumptions) is a subtask system.*

Note that if the input is such that all domain features are included, we learn a **generalized** subtask system. (In our experiments, it generalizes to all instances of a domain.)

Iteration: This process is planned to be executed iteratively as long as there are still benches that can be merged.

Regarding Work Packages 1 and 2, we already have some preliminary code and successful preliminary results for some domains. However, learning the right formula for all domains could be challenging. Because the research is carefully structured, we will gain a deeper understanding of the problem, even if the results are not as hoped.

Work Package 3: Analyzing Behaviours through Abstraction and Decomposition

Inspired by [33], the goal of this work package is to automatically decompose systems into models that represent facets of their behavior in a way that allows one to comprehend and analyze the system. Guided by the BP principles of decomposition, we will systematically dismantle the overall behavior of systems into smaller, more manageable components. This strategic breakdown will enhance each system element's conceptualization, understanding, and maintenance and set the stage for a more detailed analysis.

Complementing the decomposition process is the application of abstraction, a carefully designed mapping of system behavior to a less detailed version that empowers each model component to focus on specific system patterns in isolation. In more formal terms, the type of abstraction we seek can be achieved through a transformation that maps one regular language to another, potentially with a different alphabet but with reduced state complexity.

Imagine, for instance, the task of analyzing game logs when the rules are unfamiliar, and our objective is to learn and understand them autonomously. If we were to map all the letters representing Player 1 to "A" and all the letters representing Player 2 to "B" in the log, we would end up with a log from the language of an automaton with only two states. This language represents the alternating play pattern between the two players. The simplified automaton can be easily learned from the log, providing a component for the model we aim to construct for the game. This model, in turn, offers valuable insights into the game's structure and dynamics.

We are poised to explore and compare various techniques for identifying transformations that enable extracting valuable information from complex logs, facilitating the creation of composite models wherein each component signifies a distinct aspect of the logged behavior. Our strategies encompass:

- **Generative Layers in Neural Networks:** We plan to enhance neural networks, such as LogBERT [15], with generative layers capable of generating components for a model of the system. This augmented network aims to generate models directly from logs. To train this generative network, we will curate a dataset comprising composite models paired with logs generated from them. The training process will unfold in reverse, moving from logs to composite models.

- **Evolutionary Algorithms for Transformations:** We will generate a set of transformations and employ evolutionary algorithms [11] to construct trees that amalgamate these transformations. The fitness function driving the evolution will be the state complexity of the automaton learned through the candidate transformation.

- **Advancing Automata-Learning Techniques for Comprehending Composite Systems:** We will, for example, enhance the L^* algorithms for active learning to provide explanatory insights into bugs. Beginning with a concise log of both failed and successful tests and armed with a model of the testing framework, we will employ a modified version of the L^* algorithm. We will derive a 3-way automaton that categorizes tests into "passed," "failed," and "unknown" outcomes. Then, we will define a quantitative metric for automaton complexity and leverage solvers to identify the "simplest" explanations for a bug in the form of a regular 2-way automaton that aligns with the 3-way automaton obtained earlier.

- **Advancing Process Mining Techniques for Understanding Composite Systems:** We will, for example, introduce a new node type called "conjunction" to process trees. Each branch of these nodes represents a distinct regular language, and the subtree rooted at the conjunction node represents the intersection of these languages. We will demonstrate that process trees with this node type enable exponential size reduction while maintaining high precision and accuracy compared to process trees without this node. We will also design miners capable of learning such trees directly from logs.

Our commitment extends to the development of algorithms and proof-of-concept tools for each approach mentioned above, as well as for any additional ideas that emerge during the course of our research. By evaluating and comparing their performance, we aim to provide a diverse toolset. Recognizing the heuristic nature of this field, we acknowledge that a single method may not universally suit all purposes. Hence, our approach involves the development of multiple methods to offer users a comprehensive and adaptable toolkit. Our research trajectory pivots towards advancing the understanding and application of composite systems. We aspire to pioneer techniques for learning decomposed models, leveraging networks to decipher composed systems, and conducting a user study to assess various decomposition formalisms.

Work Package 4: Inferring Properties of Behavioral Programs from b-threads

This work package is dedicated to advancing robust methodologies and tools for the compositional analysis of reactive systems. Our primary goal is to elevate development and documentation through a comprehensive correctness proof. Harnessing the principles of BP, our approach involves analyzing individual b-threads to deduce the properties of the combined system, eliminating the need to scrutinize the extensive product space that encompasses the overall state space. This initiative embodies our overarching vision, signifying a paradigm shift in program verification methodologies.

In a previous study [16], we delved into the feasibility of expressing thread properties as logical formulas processed by an SMT solver for global property verification. From this exploration, we gained valuable insights into the simplicity and strictness of the computational model within BP, enabling reasoning about modules without the necessity for complex assume-guarantee proofs (e.g., [4, 12]) or the circular reasoning often found in less constrained paradigms. Based on this experience, as elaborated in our position paper [18], we set the following objectives for our research:

- Demonstrate the decomposability of models into modules, making the verification of module properties significantly more efficient than verifying the composite program;
- Ensure that the inherent nature of the modules allows for the formulation of meaningful properties straightforwardly, potentially implying the desired global property.

To address the first objective, we will explore methods to formulate module properties conducive to automatic reasoning systems, such as SMT solvers, which can effectively deduce the properties of the entire system. Building on existing isolated examples, including the one presented in the introduction of this proposal and an analysis demonstrating that a player in the game of Tic-Tac-Toe, employing a strategy composed of b-threads with specific properties, never loses, we aim to extend this feasibility beyond mere illustrative cases.

In the progression of this work package, our key advancement will be demonstrating the practicality of this approach in real-world scenarios, moving beyond toy examples. We will undertake controller synthesis tasks across diverse domains, encompassing robotics, traffic management, automotive systems, and cyber-physical systems. Examples include designing controllers for robot arms, traffic light control, adaptive cruise control, HVAC systems, network congestion control, drug infusion control, UAV flight, and renewable energy grid control. Each of these instances requires the creation of controllers that meet specific performance and safety specifications.

Through our investigation, we intend to showcase how BP facilitates decomposing complex systems into components that guarantee properties. We will show that these properties, in turn, enable logical solvers, such as SMT solvers, to deduce the correctness of the controller automatically. We will extend existing research such as the work on verifying response specifications in hierarchical event-based systems [7]. We will also follow current work on the verification of systems controlled by deep neural networks [34, 26].

For the second objective, our exploration will delve into methods to affirm that a given b-thread satisfies specified properties and investigate machine learning techniques to generate candidate properties with a high likelihood of both satisfaction by the b-thread and utility in the verification process. Our primary tool for confirming that a given b-thread adheres to a specified property will be model checking, encompassing both symbolic and concrete methods, leveraging the advantage that b-threads typically exhibit compact sizes.

To infer potentially beneficial properties for b-threads, we will employ generative AI models trained on a dataset we will create. This dataset will comprise b-threads accompanied by their respective properties. Acknowledging that manually generating properties for b-threads can be labor-intensive, we will employ a reverse approach: utilizing large language models such as GitHub Copilot to generate b-threads from given specifications, embedding the specification itself as the property of the b-thread. A preliminary experiment conducted in preparation for this proposal has demonstrated the feasibility of this approach.

References

[1] G. Amram, S. Maoz, and O. Pistiner. Gr(1)*: GR(1) specifications extended with existential guarantees. *Formal Aspects Comput.*, 33(4-5):729–761, 2021.

[2] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2003.

[3] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth, 1984.

[4] K. Chatterjee and T. A. Henzinger. Assume-guarantee synthesis. In O. Grumberg and M. Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 261–275, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[5] L. D. Moura and N. Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, page 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[6] R. Dechter and J. Pearl. Generalized best-first search strategies and the optimality of A*. *JACM*, 32(3):505–536, 1985.

[7] C. Disenfeld and S. Katz. Developing and verifying response specifications in hierarchical event-based systems. *LNCS Trans. Modul. Compos.*, 1:41–79, 2016.

[8] J. E. Doran and D. Michie. Experiments with the graph traverser program. *Proceedings of the Royal Society A*, 294:235–259, 1966.

[9] D. Drexler, J. Seipp, and H. Geffner. Learning sketches for decomposing planning problems into subproblems of bounded width. In *Proc. ICAPS 2022*, pages 62–70, 2022.

[10] P. Ferber, L. Cohen, J. Seipp, and T. Keller. Learning and exploiting progress states in greedy best-first search. In *Proc. IJCAI 2022*, pages 4740–4746, 2022.

[11] C. Ferreira. *Gene Expression Programming in Problem Solving*, pages 635–653. Springer London, London, 2002.

[12] C. Flanagan, S. N. Freund, and S. Qadeer. Thread-modular verification for shared-memory programs. In Daniel Le Métayer, editor, *Programming Languages and Systems*, pages 262–277, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[13] G. Francès, B. Bonet, and H. Geffner. Learning general planning policies from small examples without supervision. In *Proc. AAAI 2021*, pages 11801–11808, 2021.

[14] G. Francès, B. Bonet, and H. Geffner. Learning general policies from small examples without supervision. arXiv:2101.00692 [cs.AI], 2021.

[15] H. Guo, S. Yuan, and X. Wu. Logbert: Log anomaly detection via bert, 2021.

[16] D. Harel, A. Kantor, G. Katz, A. Marron, L. Mizrahi, and G. Weiss. On composing and proving the correctness of reactive behavior. In *2013 Proceedings of the International Conference on Embedded Software, EMSOFT 2013*, 2013 Proceedings of the International Conference on Embedded Software, EMSOFT 2013. Institute of Electrical and Electronics Engineers, 2013.

[17] D. Harel, G. Katz, R. Lampert, A. Marron, and G. Weiss. On the succinctness of idioms for concurrent programming. In Luca Aceto and D. de Frutos-Escrig, editors, *26th International Conference on Concurrency Theory, CONCUR 2015, Madrid, Spain, September 1.4, 2015*, volume 42 of *LIPIcs*, pages 85–99. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015.

[18] D. Harel, G. Katz, A. Marron, and G. Weiss. The effect of concurrent programming idioms on verification: A position paper. In *2015 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 363–369, 2015.

[19] D. Harel, A. Marron, and G. Weiss. Behavioral programming. *Commun. ACM*, 55(7):90–100, jul 2012.

[20] M. Helmert and G. Röger. How good is almost perfect? In *Proc. AAAI 2008*, pages 944–949, 2008.

[21] M. Heusner, T. Keller, and M. Helmert. Understanding the search behaviour of greedy best-first search. In *Proc. SoCS 2017*, pages 47–55, 2017.

[22] M. Heusner, T. Keller, and M. Helmert. Best-case and worst-case behavior of greedy best-first search. In *Proc. IJCAI 2018*, pages 1463–1470, 2018.

[23] J. Hoffmann and B. Nebel. The FF planning system: Fast plan generation through heuristic search. *JAIR*, 14:253–302, 2001.

[24] R. C. Holte. Common misconceptions concerning heuristic search. In *Proc. SoCS 2010*, pages 46–51, 2010.

[25] R. E. Korf, M. Reid, and S. Edelkamp. Time complexity of iterative-deepening A*. *AIJ*, 129:199–218, 2001.

[26] H. Kugler. Formal verification for natural and engineered biological systems. In *2020 Formal Methods in Computer Aided Design, FMCAD 2020, Haifa, Israel, September 21-24, 2020*, page 1. IEEE, 2020.

[27] S. Maoz and J. O. Ringert. Spectra: a specification language for reactive systems. *Softw. Syst. Model.*, 20(5):1553–1586, 2021.

[28] A. Martelli. On the complexity of admissible search algorithms. *AIJ*, 8:1–13, 1977.

[29] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins. PDDL – The Planning Domain Definition Language – Version 1.2. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, Yale University, 1998.

[30] A. Meurer, C. Smith, M. Paprocki, O. Čertík, S. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J. Moore, S. Singh, T. Rathnayake, S. Vig, B. Granger, R. M., F. Bonazzi, H. Gupta, S. Vats, F. Johansson, F. Pedregosa, M. Curry, A. Terrel, Š. Roučka, A. Saboo, I. Fernando, S. Kulal, R. Cimrman, and A. Scopatz. SymPy: symbolic computing in Python. *PeerJ Computer Science*, 3:e103, 2017.

[31] J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, 1984.

[32] N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive(1) designs. In E. Allen Emerson and Kedar S. Namjoshi, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 364–380, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[33] J. Sifakis and D. Harel. Trustworthy autonomous system development. *ACM Trans. Embed. Comput. Syst.*, 22(3):40:1–40:24, 2023.

[34] J. Tian, D. Zhi, S. Liu, P. Wang, G. Katz, and M. Zhang. Taming reachability analysis of dnn-controlled systems via abstraction-based training, 2023.

[35] C. Wilt and W. Ruml. Speedy versus greedy search. In *Proc. SoCS 2014*, pages 184–192, 2014.

[36] C. Wilt and W. Ruml. Building a heuristic for greedy search. In *Proc. SoCS 2015*, pages 131–139, 2015.

[37] C. Wilt and W. Ruml. Effective heuristics for suboptimal best-first search. *JAIR*, 57:273–306, 2016.