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 by [15] 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 [8], 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. [15] 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 [6], 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 [8], 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 [6] 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 [8], 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 [12] and 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 behavioral programming 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

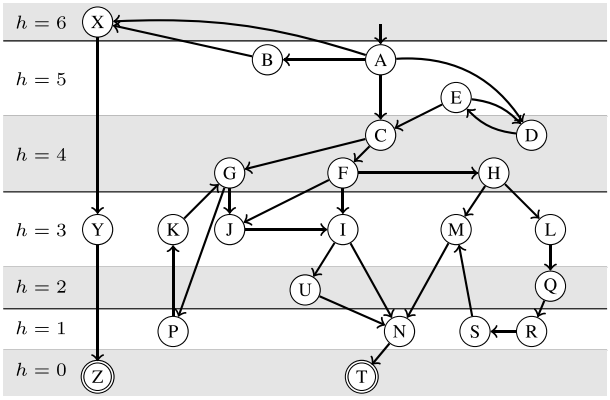

Figure 1: Search space topology from [15] for a toy domain, where the goal states are indicated by double lines.

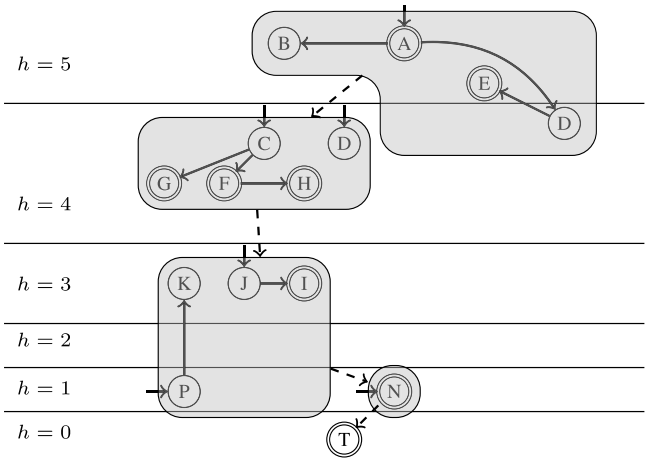

Figure 2: Search space topology from [15] for a toy domain, divided into benches connected via progress states. The progress states are indicated by double lines. Note that the progress states are actually the goal states for each bench.

behavioral threads interact within 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 behavioral programming and automatic logical reasoning.

Consider a reactive system composed of three key b-threads, each contributing to the system's overall behavior. B-thread A continuously requests an event, indicating a persistent need for a specific action. 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 [3], 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 prod-

uct space. In previous work, [11], 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 behavioral programming.

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 behavioral programming into the efficiency of different analysis tasks. By doing so, we hope to enhance the capabilities of behavioral programming, 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 behavioral programming. 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 behavioral programming 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 behavioral programming, 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) [21]. 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 $\langle pre, add, del \rangle$ with $pre, add, del \subseteq F$ and is associated with a cost $cost(a) \in \mathbb{R}_0^+$. Action a is applicable in state s if $pre \subseteq s$. Applying a in s, written as $s[\![a]\!]$, leads to the successor state $(s \setminus del) \cup add$. 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 is 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 behavioral programming.

3.3 Description Logic

Description logic (DL) is a family of knowledge representation formalisms [1] 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 $C_{p,i}(s) = \{o_i \mid \exists o_1, \ldots, o_k \text{ s.t. } p(o_1, \ldots, o_k) \in s\}$.

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 $R_{p,i,j}(s) = \{\langle o_i, o_j \rangle \mid \exists o_1, \ldots, o_k \text{ s.t. } p(o_1, \ldots, o_k) \in s\}$ 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 $(\neg X)(s) = \mathcal{O} \setminus X(s)$ resp. $(\neg X)(s) = \mathcal{O} \times \mathcal{O} \setminus X(s), (X \sqcup X')(s) = X(s) \cup X'(s),$ and$(X \sqcap X')(s) = X(s) \cap X'(s)$. respectivly.

We use the same grammar as [9]. For details, we refer to their extended paper [10].

3.4 Decision Trees

A binary decision tree is a machine learning model with a binary tree structure [2]. 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 $f(n_I) \in \{1, \ldots, F\}$ 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 [2]). 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 [16], 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 [16], 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 paper [16] the authors manually identify progress states and benches, in this proposal, our goal is to use the topology from paper [16] 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 [20, 23, 4, 19, 14, 18]. A corresponding theory for suboptimal search algorithms such as GBFS [5] has received growing attention only in the last few years [24, 25, 26, 15, 16].

The main insight of [15] 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 [16]. At this point, the high-water mark information is not needed.

In [8] they 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 [2] 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 by using the h^+ and h^{FF} heuristics [17] and 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," [11] 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

The central idea of this proposal is to deconstruct reasoning problems into their fundamental elements, such as subtasks or b-threads. This understanding will then be used to enhance algorithms and search techniques. To achieve this, we will first establish formal definitions for components, generalized components, and a system of components.

We have divided the research into specific work plans. Each of these will be assigned to Ph.D. and M.Sc. students who will be recruited and advised by us. Before delving into the specifics of each work plan, we would like to highlight the common topics addressed in this research:

- **Compositional Analysis Techniques:** We will advance the development of specific techniques for generating logical features. These features aim to describe the properties of the components and the connections between them, with the goal of deducing properties of the composed system without the need for component composition.
- **Automatic Decomposition Techniques:** We will develop innovative decomposition techniques to dissect complex systems into manageable parts. These techniques will include, but are not limited to, automata-based decompositions, subtask-based decomposition, and behavioral programming-based decompositions.
- **Learning and Exploiting Domain-Specific Patterns:** We will create new techniques for learning and exploiting common structures within specific application domains. This aspect of our research will focus on identifying patterns and structures that can be used to better understand and analyze the systems we are studying.

These contributions will form the foundation of our research and guide each work plan's direction, as elaborated below.

Work Package 1: Subtask Systems decomposition in planning problem 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 the following abstractions towards the learning phase together with heuristic and search analysis.

Generalized Subtask Systems

Definition 1 (Generalized Subtask)**.** *Let $\mathcal{D} = \langle \mathcal{P}, \mathcal{A}, \mathcal{C} \rangle$ be a planning domain and let $\mathcal{P}_G = \{p_G \mid p \in \mathcal{P}\}$ and $\mathcal{P}_I = \{p_I \mid p \in \mathcal{P}\}$. A generalized subtask $\mathcal{G}$ for $\mathcal{D}$ is a tuple $\mathcal{G} = \langle \phi_{membership}, \phi_{goal} \rangle$, where*

- *$\phi_{membership}$ is a first-order logic formula over $\mathcal{P}, \mathcal{P}_G, \mathcal{C}$;*
- *ϕ_{goal} is a first-order logic formula over $\mathcal{P}, \mathcal{P}_I, \mathcal{P}_G, \mathcal{C}$.*

Definition 2 (Subtask)**.** *Let s be a state of planning task $\langle \mathcal{O}, \mathcal{P}, \mathcal{A}, s_{\mathrm{I}}, \delta \rangle$ of domain $\mathcal{D}$, where δ is a conjunction over F, and let $\mathcal{G} = \langle \phi_{membership}, \phi_{goal} \rangle$ be a generalized subtask for $\mathcal{D}$. The (instantiated)* subtask *of s is the planning task $\Pi_{\mathcal{G}}(s) = \langle \mathcal{O}, \mathcal{P} \cup \mathcal{P}_I \cup \mathcal{P}_G, \mathcal{A}, s'_{\mathrm{I}}, \phi_{goal} \rangle$, where $s'_{\mathrm{I}} = s \cup \{f_I \mid f \in s\} \cup \{f_G \mid f \in \delta\}$.*

Definition 3 (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 $\Pi \in \mathcal{T}$, the following hold:*

- *For a state $s \in S(\Pi) \setminus S_G(\Pi)$, there is exactly one generalized subtask $\mathcal{G} = \langle \phi_{membership}, \phi_{goal} \rangle \in \mathcal{S}$ such that $s \models \phi_{membership}$, which we denote $\mathcal{S}(s)$.*
- *Let $s \in S(\Pi) \setminus S_G(\Pi)$ be a state and $\pi = \pi_0 \circ \cdots \circ \pi_n$ ($\circ$ is the concatenation of plans) be any sequence of operators such that π_0 is a plan for $\Pi_{\mathcal{S}(s)}(s)$ and π_i is a plan for $\Pi_{\mathcal{S}(s[\![\pi_0 \circ \cdots \circ \pi_{i-1}]\!])}(s[\![\pi_0 \circ \cdots \circ \pi_{i-1}]\!])$. Then, π is a (global) s-plan.*

If these properties hold for each task of $\mathcal{D}$, we say that $\mathcal{S}$ is a generalized subtask system*.*

A generalized subtask system can be used to decompose a planning task by identifying the bench for the current state s (starting with s_{I}), constructing and solving the corresponding bench task, and continuing with the resulting goal state until a (global) goal is reached.

Finally, we intend to show that,

Theorem 1. *A generalized bench transition system(plus some restrictions, e.g., perfect data classification) is a subtask system.*

Exploitation: Bench Walking Baader et al. [1] showed that any description logic (DL) concept/role can be converted to first-order logic (FOL). Any FOL is easily written in PDDL. To support this end, one of the side effects of this research is to support the translation from DL to PDDL. This is especially necessary to switch from the learned "exit" progress states represented as DL formulas into subgoals represented as PDDL, which provide the syntax and semantics needed in Fast Downward planning systems [13].

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. [15] 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
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.

As a result, for each input task Π_i, we have constructed a BTS $\mathfrak{B}_i = \langle V_i, E_i \rangle$, where V_i is a partition over $S_i \setminus S_G$ and E_i are the edges between the benches. Let $\mathfrak{B} = \langle V, E \rangle$ such that $V = \bigcup_i V_i$, $E = \bigcup_i E_i$, and $S = \bigcup_i S_i$ are all goal state S.

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.

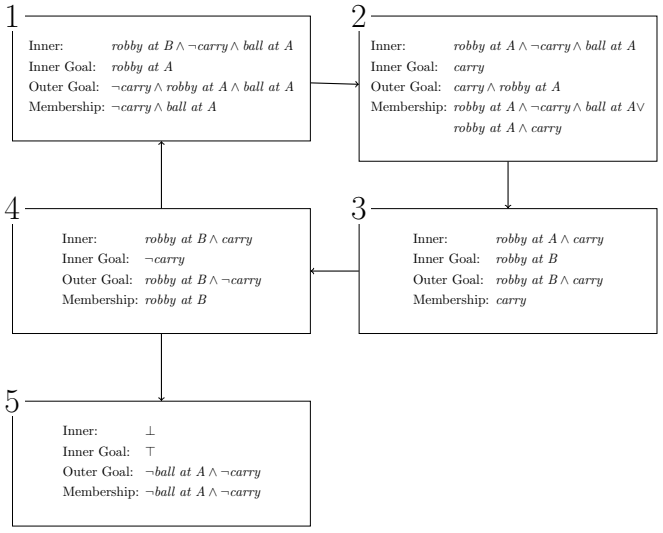

Figure 3: GBTS for a toy domain, a gripper with one arm that needs to move balls between rooms using the h^* heuristic. In each box, a formula representing the relevant states is presented.

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 [8], we train a decision tree on those samples, extract a formula in disjunctive normal form from the tree, and simplify it using SymPy [22].

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 2. *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: Decomposing Systems through Abstraction and Decomposition

This work package is intricately crafted to unravel the complexities embedded within computer systems, human-made processes, and natural phenomena. Our overarching objective is to forge sophisticated methodologies that autonomously construct models from data, thereby empowering a profound understanding and analysis of the intricate behaviors inherent in these systems.

Guided by the principles of decomposition, we systematically dismantle the overall behavior of systems into smaller, more manageable components. This strategic breakdown not only enhances the conceptualization, understanding, and maintenance of each individual system element but also sets the stage for a more detailed analysis. Complementing the decomposition process is the application of abstraction, a meticulous mapping of system behavior to a less detailed version within labeled transition systems (LTS). This abstraction empowers each model component to focus on specific system patterns in isolation, significantly improving overall comprehensibility.

At the heart of our proposed solution lie transformations—a mechanism designed to represent the behavior patterns of the system in a succinct and isolated manner. Our aim is to identify a set of transformations that is strongly equivalent to the language describing the system. This set may include an identity transformation singleton or a collection of intersection-based transformations, each weakly equivalent by definition. The optimal set of transformations ensures that the corresponding automata exhibit simplicity, gauged through various criteria such as state and transition counts.

An initial validation of the approach: To validate our methodology, we embarked on an exploration of automata-based transformations using Tic-Tac-Toe (TTT) as a model. Despite challenges in applying the RPNI algorithm to construct a deterministic finite automaton (DFA) tailored to legal TTT games, we successfully devised three specialized transformations aligning with TTT's rules. These transformations resulted in 26 DFAs,

providing an interpretable representation of TTT and highlighting the impact of automata-based structures on explainability and expressiveness. Future explorations will delve into advanced automata-related structures, including context-oriented behavioral programming [7], to elevate our modeling capabilities for intricate patterns.

Our ongoing research trajectory pivots towards advancing the understanding and application of decomposed systems. We aspire to pioneer techniques for learning decomposed systems, leveraging networks to decipher composed systems, and conducting a user study to assess various decomposition formalisms. Building on the observed success with Tic-Tac-Toe (TTT), we plan to integrate solvers and logical engines to enhance adaptability. Additionally, we are unwaveringly committed to developing a graphical modeling language for decomposed systems, refining existing formalisms through user studies. This holistic approach not only contributes significantly to the theoretical foundations of decomposed systems but also holds practical promise in system modeling and understanding.

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

This work package spearheads the development of robust methodologies and tools to compose reactive systems seamlessly. Our central objective is to augment development and documentation with comprehensive correctness proof. Our approach harnesses behavioral programming (BP) principles to analyze individual b-threads and deduce the properties of the combined system from these properties, effectively bypassing the need to scrutinize the product space encompassing the overall state space of the system. Our overarching vision entails a paradigm shift in program verification methodologies.

Example: Consider a b-program with three b-threads. Each b-thread exhibits distinct behaviors related to the "ping" event: the first blocks "ping" when time is not divisible by p, the second when not divisible by q, and the third consistently requests "ping". Leveraging an existing BP model-checker, we can verify individual b-thread properties, ensuring the correct implementation of these properties. Our paradigm shift lies in employing a deduction engine to deduce system-wide behavior. By combining the conditions from the first two b-threads, we deduce that "ping" occurs only when time is divisible by $p \cdot q$, obviating the need to scrutinize the entire state space. This innovative methodology achieves an exponential improvement in verification time, reducing it from $p \cdot q$ to $p + q$ states.

The concept of verifying behavioral programs compositionally was introduced in the work by Harel et al. [**?**]. In this study, thread properties are expressed as logical formulas and then processed by an SMT solver for the verification of global properties. An interesting insight arising from this research is that, within a simple and strict computational model, it is possible to reason about modules without relying on complex assume-guarantee proofs (e.g., [**?** **?**]) or the circular reasoning often required in less constrained paradigms. This suggests that instances like these support the idea that a straightforward model can fulfill the condition that modules allow for the formulation of meaningful properties, which may later imply the desired global property. The additional requirement, stating that programs can be decomposed into modules, with the verification of module properties being substantially more economical than verifying the composite program, is met due to the linear nature of model-checking in relation to program size. This implies the need for modules to be exponentially smaller than the composite program in terms of the number of states. While concurrency inherently enables the creation of

smaller modules, a relevant research question arises: does the limited form of concurrency provided by the BP synchronization method, along with its request, wait-for, and block idioms, suffice for this purpose?

Our method is poised to demonstrate the automation and streamlining of program verification by strategically amalgamating the properties of individual modules. These modules, intricately specified and rigorously verified in isolation, will undergo a harmonization process with application-independent specifications. This synthesis will encompass both BP semantics and general theories, engendering a synergistic effect that exponentially accelerates the verification process. This efficiency surpasses the traditional model-checking approaches for composite applications, signifying a leap forward in verification methodology.

A fundamental dimension of our work involves accentuating the inherent value of formalizing properties for independent modules. Beyond their role in facilitating correctness proofs, this formalization process stands as an invaluable source of documentation for future development endeavors. Our commitment is steadfastly aligned with the broader industry objective of instating formal correctness proofs as a standard practice in the development of reactive systems. Our vision extends beyond this work package, aspiring for widespread adoption by programmers across the industry. Through this concerted effort, we envision a future where formal proofs of correctness seamlessly integrate into the very fabric of reactive system development, championed and embraced by programmers at large.

References

[1] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2003.

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

[3] Leonardo De Moura and Nikolaj 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.

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

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

[6] *Dominik Drexler, Jendrik Seipp, and Hector Geffner. Learning sketches for decomposing planning problems into subproblems of bounded width. In *Proc. ICAPS 2022*, pages 62–70, 2022.

[7] Achiya Elyasaf. Context-Oriented Behavioral Programming. *Information and Software Technology*, 133:106504, May 2021. Publisher: Elsevier BV.

[8] Patrick Ferber, Liat Cohen, Jendrik Seipp, and Thomas Keller. Learning and exploiting progress states in greedy best-first search. In *Proc. IJCAI 2022*, pages 4740–4746, 2022.

[9] Guillem Francès, Blai Bonet, and Hector Geffner. Learning general planning policies from small examples without supervision. In *Proc. AAAI 2021*, pages 11801–11808, 2021.

[10] Guillem Francès, Blai Bonet, and Hector Geffner. Learning general policies from small examples without supervision. arXiv:2101.00692 [cs.AI], 2021.

[11] David Harel, Guy Katz, Robby Lampert, Assaf Marron, and Gera Weiss. On the succinctness of idioms for concurrent programming. In Luca Aceto and David 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.

[12] David Harel, Assaf Marron, and Gera Weiss. Behavioral programming. *Commun. ACM*, 55(7):90–100, jul 2012.

[13] Malte Helmert. The Fast Downward planning system. *JAIR*, 26:191–246, 2006.

[14] Malte Helmert and Gabriele Röger. How good is almost perfect? In *Proc. AAAI 2008*, pages 944–949, 2008.

[15] *Manuel Heusner, Thomas Keller, and Malte Helmert. Understanding the search behaviour of greedy best-first search. In *Proc. SoCS 2017*, pages 47–55, 2017.

[16] Manuel Heusner, Thomas Keller, and Malte Helmert. Best-case and worst-case behavior of greedy best-first search. In *Proc. IJCAI 2018*, pages 1463–1470, 2018.

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

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

[19] Richard E. Korf, Michael Reid, and Stefan Edelkamp. Time complexity of iterative-deepening A*. *AIJ*, 129:199–218, 2001.

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

[21] Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David 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.

[22] Aaron Meurer, Christopher Smith, Mateusz Paprocki, Ondřej Čertík, Sergey Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian Granger, Richard Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew Curry, Andy Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. SymPy: symbolic computing in Python. *PeerJ Computer Science*, 3:e103, 2017.

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

[24] Christopher Wilt and Wheeler Ruml. Speedy versus greedy search. In *Proc. SoCS 2014*, pages 184–192, 2014.

[25] Christopher Wilt and Wheeler Ruml. Building a heuristic for greedy search. In *Proc. SoCS 2015*, pages 131–139, 2015.

[26] Christopher Wilt and Wheeler Ruml. Effective heuristics for suboptimal best-first search. *JAIR*, 57:273–306, 2016.