Rewrite-Based Decomposition of Signal Temporal Logic Specifications

Kevin Leahy[1], Makai Mann[1], and Cristian-Ioan Vasile[2]

[1] MIT Lincoln Laboratory, 244 Wood St, Lexington, Massachusetts 02421
{kevin.leahy,makai.mann}@ll.mit.edu
[2] Lehigh University, 27 Memorial Dr W, Bethlehem, PA 18015
cvasile@lehigh.edu

Abstract. Multi-agent coordination under Signal Temporal Logic (STL) specifications is an exciting approach for accomplishing complex temporal missions with safety requirements. Despite significant progress, these approaches still suffer from scalability limitations. Decomposition into subspecifications and corresponding subteams of agents provides a way to reduce computation and leverage modern parallel computing architectures. In this paper, we propose a rewrite-based approach for jointly decomposing an STL specification and team of agents. We provide a set of formula transformations that facilitate decomposition. Furthermore, we cast those transformations as a rewriting system and prove that it is convergent. Next, we develop an algorithm for efficiently exploring and ranking rewritten formulae as decomposition candidates, and show how to decompose the best candidate. Finally, we compare to previous work on decomposing specifications for multi-agent planning problems, and provide computing and energy grid case studies.

1 Introduction

Coordination and control of multi-agent systems from high-level specifications is a challenging and active area of research. As with many areas of formal methods and multi-agent systems, scalability is often a limiting factor. Ideally, an operator would provide a single global specification for a large team of multi-agent systems, and the system would assign tasks and roles accordingly and synthesize a plan and controllers. Here, we aim to formalize a method for analyzing a signal temporal logic (STL) specification for a principled approach to jointly decompose and distribute the specification among a team of agents.

Most STL work in multi-agent systems assumes either centralized control from a global specification [15,11,4] or decentralized control from local specifications [16,17]. In contrast, several methods for multi-agent planning with linear temporal logic (LTL) specifications have identified methods for automatically

© 2022 Massachusetts Institute of Technology. The NASA University Leadership Initiative (grant #80NSSC20M0163) provided funds to assist the authors with their research, but this article solely reflects the opinions and conclusions of its authors and not any NASA entity.

decomposing a specification into sub-specifications and assigning agents or sub-teams of agents to execute those sub-specifications [7,21,2,23].

Unlike LTL, STL has the advantage of specifying concrete timing requirements over continuous, real-valued signals, but there has been comparatively little work focused on decomposition of STL. [6] focused on decomposing an STL formula given an *a priori* set of disjoint sub-teams. In this work, we decompose the formula and team jointly, in an attempt to achieve a task-based set of sub-teams. We take inspiration from [14], but our approach is based on an abstract reduction system, providing guarantees on its convergence. We also perform formula transformation and assignment as two separate stages, reducing the search space of the assignment problem to those that are feasible for a given transformation. Additionally, our approach works for STL in general, whereas [14] focuses only on a fragment of STL. Another closely related work is [22], which looks at a multi-agent fragment of STL and assigns sub-formulae to individual agents. The assignment is an *implicit* part of their synthesis framework. Here, we focus only on an *explicit* assignment and decomposition, and we do not consider the synthesis problem. The method presented in this work could be used as a pre-processing step for their proposed synthesis and motion planning work, as well as most other existing multi-agent STL synthesis approaches.

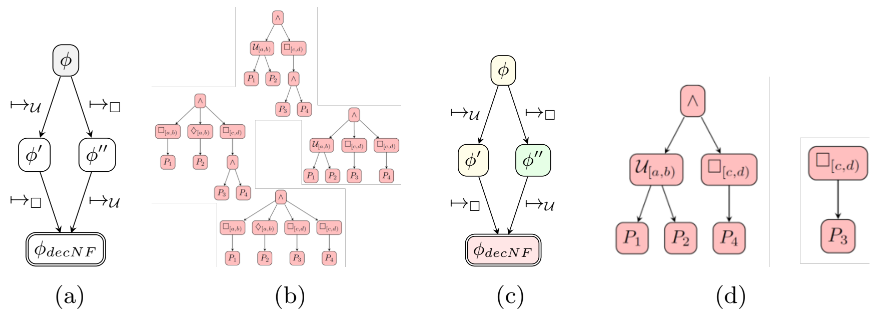


Fig. 1: Decomposition framework: build a DAG of all possible formula rewrites (1a, Sec. 3); build an AST for each resulting formula (1b, Sec. 4); score each node in the DAG according to its AST (1c, Sec. 4.3); and select best node and evaluate its decomposed specification (1d, Sec. 4.2).

The main contribution of this paper is an algorithm for simultaneous decomposition of STL formulae and heterogeneous agent subteam assignment consisting of 1) a rewriting system for reasoning about changes to STL formulae, with proof of termination and confluence; 2) a normal form of decomposed STL formulae that provides bounds on the ability to decompose a formula; and 3) an optimization approach for selecting the best decomposition and assignment based on a directed acyclic graph (DAG) constructed by the rewriting system.

2 Background and Problem Definition

In this work, we focus on requirements in the form of Signal Temporal Logic (STL) specifications [18]. The syntax of STL is given in Backus-Naur form as

$$\phi := \top \mid P \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \mathcal{U}_{[a,b)} \phi_2 \ , \tag{1}$$

where ϕ, ϕ_1, and ϕ_2 are STL formulae; $\top$ is the symbol for logical *true*; P is a predicate of the form $\pi(\mathbf{x}(t)) \geq c$ for $\mathbf{x} \in \mathbb{R} \to \mathbb{R}^m$, $\pi \in \mathbb{R}^m \to \mathbb{R}$, and $c \in \mathbb{R}$; $\neg$ and $\wedge$ are Boolean negation and conjunction; and $\mathcal{U}_{[a,b)}$ is the temporal operator Until, with $a, b \in \mathbb{R}$ and $a \leq b$. Other operators $\vee$ (disjunction, $\neg(\neg\phi_1 \wedge \neg\phi_2)$), $\Diamond$ (finally, $\top\mathcal{U}_{[a,b)}\phi$), and $\Box$ (globally, $\neg\Diamond_{[a,b)}\neg\phi$) can be defined from the other operators. We use the notation $pred(\phi)$ to denote all predicates in ϕ, and $conj(\phi)$ for all the *top-level* conjuncts of formula ϕ.

Example 1. Let P_i be STL predicates, and $\phi \coloneqq P_1\mathcal{U}_{[a,b)}P_2 \wedge \Box_{[c,d)}(P_3 \wedge P_4)$:

$$\begin{aligned} pred(\phi) &\coloneqq \{P_1, P_2, P_3, P_4\} \\ conj(\phi) &\coloneqq \{P_1\mathcal{U}_{[a,b)}P_2, \Box_{[c,d)}(P_3 \wedge P_4)\} \end{aligned}$$

Note that although $P_3 \wedge P_4$ contains a conjunction, it is not at the *top-level.*

The semantics of STL with respect to a signal $\mathbf{x}$ at time t are defined as

$$\begin{aligned} (\mathbf{x}, t) \models \pi(\mathbf{x}(t)) \geq c \quad &\Leftrightarrow \pi(\mathbf{x}(t)) \geq c \\ (\mathbf{x}, t) \models \neg\phi \quad &\Leftrightarrow (\mathbf{x}, t) \not\models \phi \\ (\mathbf{x}, t) \models \phi_1 \wedge \phi_2 \quad &\Leftrightarrow (\mathbf{x}, t) \models \phi_1 \text{ and } (\mathbf{x}, t) \models \phi_2 \\ (\mathbf{x}, t) \models \phi_1\mathcal{U}_{[a,b)}\phi_2 \quad &\Leftrightarrow \exists t' \in [t+a, t+b] s.t. (\mathbf{x}, t') \models \phi_2 \text{ and} \\ &\qquad \forall t'' \in [0, t'](\mathbf{x}, t'') \models \phi_1 \ . \end{aligned} \tag{2}$$

In addition to the semantics defined above, STL also has the notion of quantitative semantics or *robustness degree*, ρ. The robustness of a signal $\mathbf{x}$ at time t with respect to formula ϕ is defined as [9]

$$\begin{aligned} \rho(\mathbf{x}, t, \pi(\mathbf{x}(t)) \geq c) \quad &\coloneqq \pi(\mathbf{x}(t)) - c \\ \rho(\mathbf{x}, t, \neg\phi) \quad &\coloneqq -\rho(\mathbf{x}, t, \phi) \\ \rho(\mathbf{x}, t, \phi_1 \wedge \phi_2) \quad &\coloneqq \min(\rho(\mathbf{x}, t, \phi_1), \rho(\mathbf{x}, t, \phi_2)) \\ \rho(\mathbf{x}, t, \phi_1\mathcal{U}_{[a,b)}\phi_2) \quad &\coloneqq \max_{t' \in [t+a, t+b]}(\rho(\mathbf{x}, t', \phi_2), \min_{t'' \in [t, t']} \rho(\mathbf{x}, t'', \phi_1)) \end{aligned} \tag{3}$$

The *horizon* of an STL formula is the maximum execution time before the satisfiability of the specification can be determined [8]. The formula horizon, *hzn*, is defined recursively as:

$$\begin{aligned} hzn(\pi(\mathbf{x}(t)) \geq c) \quad &\coloneqq 0 \\ hzn(\neg\phi) \quad &\coloneqq hzn(\phi) \\ hzn(\phi_1 \circ \phi_2) \quad &\coloneqq max(hzn(\phi_1), hzn(\phi_2)) \text{ for } \circ \in \{\wedge, \vee\} \\ hzn(\phi_1\mathcal{U}_{[a,b)}\phi_2) \quad &\coloneqq max(hzn(\phi_1) + b - 1, hzn(\phi_2) + b) \end{aligned} \tag{4}$$

Definition 1 (Agent) *An* agent *is a tuple $A = (x(t), u)$, where $x(t) \in \mathbb{R}^n$ is its n-dimensional state at time t, and $u \in \mathbb{R}^n$ serves as an element-wise upper bound on $x(t)$. The lower bound is assumed to be an n-dimensional zero vector.*

The upper- and lower-bounds on $x(t)$ do not change with time. Rather, they bound $x(t)$ across all time. For convenience, we often drop the explicit dependence of $x(t)$ on t and simply write x. We denote the signal for a team of agents as $\mathbf{x} \in \mathbb{R}^m$. This can be obtained via concatenation, summation or other operation over individual agent signals. We assume this can be done but are agnostic to how. We use the term *agent* to be consistent with the related robotics literature, but agents represent any entities or processes that can be controlled separately. We say that an agent "services" a predicate if it is responsible for maintaining a signal that satisfies that predicate (or contributes to its satisfaction).

For a team of agents indexed by set J, we denote the j^{th} agent as A_j, where $j \in J$. If two agents have the same upper bound u, we consider them to belong to an equivalence class g_u. For a team of agents, we denote the set of all such equivalence classes as G.

Given a (sub)team of agents $\mathcal{A} = \{A_j\}_{j \in J}$, we define the robustness *upper bound*, ρ_{ub}, recursively as:

$$\begin{aligned}
\rho_{ub}(\pi(\mathbf{x}(t)) \geq c), \mathcal{A}) &:= (\Sigma_{a \in \mathcal{A}} \pi(a.u)) - c \\
\rho_{ub}(\neg\phi, \mathcal{A}) &:= -\rho_{ub}(\phi, \mathcal{A}) \\
\rho_{ub}(\phi_1 \wedge \phi_2, \mathcal{A}) &:= min(\rho_{ub}(\phi_1, \mathcal{A}), \rho_{ub}(\phi_2, \mathcal{A})) \\
\rho_{ub}(\phi_1 \vee \phi_2, \mathcal{A}) &:= max(\rho_{ub}(\phi_1, \mathcal{A}), \rho_{ub}(\phi_2, \mathcal{A})) \\
\rho_{ub}(\phi_1 \mathcal{U}_{[a,b)} \phi_2, \mathcal{A}) &:= min(\rho_{ub}(\phi_1, \mathcal{A}), \rho_{ub}(\phi_2, \mathcal{A}))
\end{aligned} \tag{5}$$

This is the typical robustness definition evaluated over the agent upper bounds for every agent in the given team. Note that it no longer depends on $\mathbf{x}$ or t. If the robustness upper bound is negative, there does not exist a synthesized plan for agents $\mathcal{A}$ that satisfies the formula.

Example 2. To illustrate our notion of agents, we consider an example from computing. For a large computing cluster, each compute node can be modeled as an agent, with the state x capturing its resource utilization between its CPU, GPU, and RAM. Different predicates in an STL formula ϕ might request different combinations of resources. For a node with 16 CPU cores, 800 GPU cores, and 64 GB of RAM, its state is 3-dimensional with upper bound $u = \{16, 800, 64\}$ representing 100% utilization. Let the class of this agent be denoted g_1. Any other agent with exactly the same values of u would also belong to g_1, otherwise, the agent would belong to a separate class. The team signal, $\mathbf{x}$, is a concatenation of the agent signals.

Assumption 1 *We assume the existence of a synthesis method, `Synth`. Given a team of agents J and an STL specification ϕ, `Synth`(J, ϕ) synthesizes a plan for the agent(s) J to satisfy ϕ, if such a plan exists.*

Synthesis is not the focus of this work, and there are many synthesis tools available depending on the specific problem under consideration. Assumption 1 simply states that a user of our proposed method has an appropriate synthesis tool available.

We are now ready to state the problem under consideration. We seek a method for decomposing a synthesis problem, consisting of a team of agents and an STL specification, into a set of smaller, disjoint subproblems that can each be solved independently. The goal of this decomposition is to achieve a faster solution than solving the original problem, without rendering the problem infeasible in the process of decomposing it. However, there is no known method for determining the feasibility of a synthesis problem without running the synthesis procedure, which is expensive. Therefore we rely on robustness upper bound, which is a necessary condition for feasibility.

Problem 1. Given a team of agents $\{A_j\}_{j \in J}$ and an STL formula ϕ, find a team partition R and a set of formulae $\{\phi_r\}_{r \in R}$ such that

1. ϕ is satisfied if each subteam $r \in R$ satisfies its specification ϕ_r;
2. Solving the set of synthesis problems $\texttt{Synth}(r, \phi_r)$, including the time to decompose, is faster than $\texttt{Synth}(J, \phi)$; and
3. Robustness upper bound for all teams is non-negative.

Assumption 2 *We assume that the original synthesis problem has a solution.*

The focus of this work is on decomposition of a problem into subproblems whose solution guarantees solution of the original problem. For our analysis, Assumption 2 restricts us to feasible problems, since every infeasible problem will yield at least one infeasible subproblem.

Our approach is depicted in Fig. 1. The first step generates various transformations of the original formula that are easier to decompose. Next, the technique builds an abstract syntax tree (AST) for each of these formulae, efficiently explores the possible transformations and scores each rewritten formula. Finally, it selects the best node and decomposes the specification into subspecifications and associated subteams. We cover each of these steps in the following sections.

3 An STL Rewriting System

We start by describing a technique for modifying STL formulae to be more amenable to decomposition, while still guaranteeing satisfaction of the original formula. We accomplish this by developing a rewriting system for STL. Rewriting operates on abstract syntax trees (ASTs). Every STL formula can be represented as an AST with each node representing an operator or predicate. See Dfn. 7 in the Supplementary Material for a formal definition.

Formulae consisting of top-level conjunctions are the easiest to decompose. Top-level conjuncts are represented in an AST as the children of the root node, which is a conjunction operator. Logically, if each top-level conjunct is satisfied, then the entire formula is satisfied. There is no logical dependence between any of the top-level conjuncts. In this section, we provide three formula transformations that can add top-level conjuncts to facilitate decomposition. Not all of these transformations produce an equisatisfiable formula. However, we ensure that the transformation only strengthens the formula. Let ϕ be an STL formula and τ a transformation operator, then we only consider transformations such that $\tau(\phi) \models \phi$. This ensures that any plan found for the transformed formula is guaranteed to satisfy the original formula.

Definition 2 (Rewriting System) *A rewriting system is a tuple, $(T, \rightarrow)$, where T is a set of* terms*, and $\rightarrow \subseteq T \times T$ is a rewriting relation on T. If the terms y and z are in $\rightarrow$, we write $y \rightarrow z$.*

We now define our rewriting system for STL, $(\mathbb{S}, \rightarrow_{STL})$, where $\mathbb{S}$ is the set of all STL formulae, and $\rightarrow_{STL} := \{\mapsto_{\Box}, \mapsto_{\Diamond}, \mapsto_{\mathcal{U}}\}$ is a collection of rewrite rules defined below. Let ϕ_1 and ϕ_2 be STL formulae, and $a \leq b$ be real-valued time instances. We consider the following rewriting rules in $\rightarrow_{STL}$:

$$\Box_{[a,b)}(\phi_1 \wedge \phi_2) \mapsto_{\Box} \Box_{[a,b)}(\phi_1) \wedge \Box_{[a,b)}(\phi_2) \qquad \text{(split-globally)}$$

$$\Diamond_{[a,b)}(\phi_1 \wedge \phi_2) \mapsto_{\Diamond} \Box_{[a,b)}(\phi_1) \wedge \Box_{[a,b)}(\phi_2) \qquad \text{(split-finally)}$$

$$\phi_1 \mathcal{U}_{[a,b)} \phi_2 \mapsto_{\mathcal{U}} \Box_{[0,b)}(\phi_1) \wedge \Diamond_{[a,b)}(\phi_2) \qquad \text{(split-until)}$$

Of these rewriting rules, only (split-globally) produces an equisatisfiable formula. The other two entail the original formula as required, but are not satisfied by every trace that satisfies the original formula.

Remark 1. Our (split-finally) transformation is the most conservative of several possible choices for splitting $\Diamond$ over a conjunction. Both $\Diamond_{[a,b)}(\phi_1) \wedge \Box_{[a,b)}(\phi_2)$ and $\Box_{[a,b)}(\phi_1) \wedge \Diamond_{[a,b)}(\phi_2)$ also entail the original formula. We choose the symmetric option for simplicity of presentation.

Remark 2. We include two other transformations that produce a formula equisatisfiable to the input. If two like temporal operators appear next to each other in the formula, we adjust their time bounds accordingly. That is, $\Box_{[a,b)}\Box_{[c,d)}\phi \mapsto_{\Box\Box} \Box_{[a+c,b+d)}\phi$ and likewise $\Diamond_{[a,b)}\Diamond_{[c,d)}\phi \mapsto_{\Diamond\Diamond} \Diamond_{[a+c,b+d)}\phi$. We omit these rewrite rules from our presentation for simplicity, but all subsequent proofs and analyses hold for these rewrites as well.

Theorem 1. *The rewriting system $(\mathbb{S}, \rightarrow_{STL})$ terminates.*

Proof (Sketch). By Lemma 2.3.3 of [1], a finitely branching rewriting system $(T, \rightarrow)$ terminates if there exists a monotone embedding φ from $(T, \rightarrow)$ into $(\mathbb{N}, >)$. In our case, φ has two components – the sum of distances of conjuncts and until operators from the root, and the number of occurrences of conjunction and until operators in the formula. Namely,

$$\varphi = \sum_{i=1}^{n_{\wedge,\mathcal{U}}} d_i + (n_{\wedge,\mathcal{U}} + 1) n_{\mathcal{U}} \; , \tag{6}$$

where $n_\wedge$ and $n_\mathcal{U}$ are the number of conjunction and until operators appearing in the formula, $n_{\wedge,\mathcal{U}} = n_\wedge + n_\mathcal{U}$, and d_i is the distance of the i^{th} conjunction or until from the root in the formula AST. For all replacement rules we consider, φ is a monotone embedding into $(\mathbb{N}, >)$, and our rewriting system terminates. $\square$

Let $\rightarrow$ be an arbitrary reduction system containing a nonempty set of reduction mappings $\mapsto_i$, and $\xrightarrow{*}$ be its reflexive, transitive closure. Terms y and w are *joinable*, denoted $y \downarrow w$, *iff* there is a z such that $y \xrightarrow{*} z \xleftarrow{*} w$.

Definition 3 (Confluence) *A reduction system is* confluent *if* $\forall y \,.\, w_1 \xleftarrow{*} y \xrightarrow{*} w_2 \implies w_1 \downarrow w_2$.

Theorem 2. *The reduction system generated by* (split-globally), (split-finally), *and* (split-until) *is confluent.*[3]

Proof (Sketch). To prove confluence, we break our formula rewriting reduction system into three independent reduction systems:

1. $\rightarrow_\square := \{\mapsto_\square\}$: reduction system for (split-globally)
2. $\rightarrow_\diamond := \{\mapsto_\diamond\}$: reduction system for (split-finally)
3. $\rightarrow_\mathcal{U} := \{\mapsto_\mathcal{U}\}$: reduction system for (split-until)

We prove confluence by proving that each individual reduction system is confluent and commutative, then building up to our full reduction system, $\rightarrow_{STL} := \{\mapsto_\square, \mapsto_\diamond, \mapsto_\mathcal{U}\}$. Each reduction system alone is trivially confluent. We now look at combinations of reduction systems.

$\rightarrow_\square$ and $\rightarrow_\diamond$ act on different temporal operators. Therefore each can be applied independently, making local changes to non-overlapping regions of the AST (see Sec. 4), and the final ASTs are the same. This implies that $\rightarrow_{\square\diamond} := \rightarrow_\square \cup \rightarrow_\diamond$ is confluent. The same logic applies to $\rightarrow_{\square\diamond}$ and $\rightarrow_\mathcal{U}$, and therefore $\rightarrow := \rightarrow_{\square\diamond} \cup \rightarrow_\mathcal{U} = \rightarrow_\square \cup \rightarrow_\diamond \cup \rightarrow_\mathcal{U}$ is also confluent. $\square$

Complete proofs for Theorems 1 and 2 can be found in the Supplementary Materials. Because our STL formula rewriting system is terminating and confluent, it is *convergent*. This implies that any STL formula can be reduced to a unique normal form through the application of our rewriting rules. We will call this form *decomposition normal form* (decNF). Importantly, satisfaction of a formula in decNF form implies satisfaction of the original formula, but not vice-versa. The number of top-level conjuncts in an STL formula in decNF is an upper bound on the number of subteams our method will produce.

[3] We assume that there is a global subterm ordering that puts equivalent formulae in a normal form, i.e., $b \wedge a \rightarrow a \wedge b$ so that $a \wedge b$ and $b \wedge a$ are known to be identical.

3.1 Formula Rewrite DAG

Given $(\mathbb{S}, \rightarrow_{STL})$ and an STL formula, we can consider all possible formulae obtained by repeated application of the rewriting rules to subformulae.

Definition 4 (Rewrite DAG) *A Formula Rewrite DAG, $G \coloneqq \langle \Phi, E \rangle$, is a directed acyclic graph where each node in Φ is an STL formula and each directed edge in E goes from ϕ_1 to ϕ_2 such that $\phi_1 \rightarrow_{STL} \phi_2$.*

We denote the Formula Rewrite DAG for a formula, ϕ, as $G(\phi)$. For all ϕ, $G(\phi)$ has a single source node (in-degree of 0) corresponding to the original formula, ϕ, and a single sink node (out-degree of 0) corresponding to the decNF form of ϕ.

Example 3. Let ϕ be the formula from Example 1. One edge in $G(\phi)$ would connect ϕ (the root) to $P_1 \mathcal{U}_{[a,b)} P_2 \wedge \Box_{[c,d)} P_3 \wedge \Box_{[c,d)} P_4$. This edge would be tagged with the transformation $\rightarrow_{\Box}$ and the conjunct it was applied to, $\Box_{[c,d)}(P_3 \wedge P_4)$. Fig. 2 shows the complete rewrite DAG.

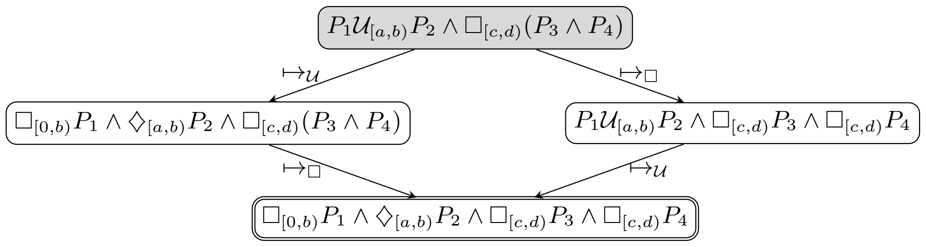

$P_1 \mathcal{U}_{[a,b)} P_2 \wedge \Box_{[c,d)} (P_3 \wedge P_4)$
$\mapsto_{\mathcal{U}}$
$\mapsto_{\Box}$
$\Box_{[0,b)} P_1 \wedge \Diamond_{[a,b)} P_2 \wedge \Box_{[c,d)} (P_3 \wedge P_4)$
$P_1 \mathcal{U}_{[a,b)} P_2 \wedge \Box_{[c,d)} P_3 \wedge \Box_{[c,d)} P_4$
$\mapsto_{\Box}$
$\mapsto_{\mathcal{U}}$
$\Box_{[0,b)} P_1 \wedge \Diamond_{[a,b)} P_2 \wedge \Box_{[c,d)} P_3 \wedge \Box_{[c,d)} P_4$

Fig. 2: Example of a rewrite DAG illustrating confluence and termination from the spec in Example 1. P_1, P_2, P_3, and P_4 represent STL predicates. a, b, c, and d are real-valued time bounds. The initial (root) formula is in the top gray box. Decomposition normal form is indicated by the double rectangle. Rewrite operations are indicated on the edges between formulae in the DAG.

Theorem 3. *Constructing a rewrite DAG by enumerating all possible formula transformations terminates with a finite graph.*

Proof. This follows directly from Theorem 1.

4 Decomposing STL

Having created a rewrite DAG as described in Sec. 3 above, we now describe how to analyze the candidate formula at each node in the DAG. For an STL formula at a given node in the DAG, we wish to assign agents to that formula in a way that is amenable to decomposition.

4.1 Agent Assignments

We now define agent assignments, followed by their use in STL decomposition.

Definition 5 (Agent Assignment) *An agent assignment for STL formula ϕ is a mapping $\alpha : conj(\phi) \to 2^J$.*

The mapping records agent assignments to top-level conjuncts of the specification. These are the children of the root AST node. The assignment is extended to all AST nodes by adopting the agent assignment of parent nodes. The root node agent assignment is defined to be all agents.

4.2 Decomposition

An agent assignment for a given STL specification induces a team partition R and a set of subspecifications, $\{\phi_r\}_{r \in R}$. Recall that J is the set of agent indices, thus R is a set of (nonempty) subsets that partition the agents. Formally, each element of R is a subset $r \subseteq J$ such that $\forall r \in R \,.\, |r| > 0$ and $\forall r_i, r_j \in R \,.\, i \neq j \to r_i \cap r_j = \emptyset$.

Given an agent assignment, α, R is computed as the largest valid partition such that $\forall r \in R \; \forall c \in conj(\phi) \,.\, \alpha(c) \subseteq r \vee \alpha(c) \cap r = \emptyset$. Intuitively, this is computed by starting with agent assignments for each top-level conjunct, and combining any conjuncts that have overlapping agent assignments. Let $match(r, \phi) \coloneqq \{c | c \in conj(\phi) \wedge \alpha(c) \in r\}$. This denotes all top-level conjuncts associated with agent partition r. For each $r \in R$, there is a corresponding subspecification, $\phi_r \coloneqq \bigwedge_{c \in match(r,\phi)} c$. Let a decomposition, $\mathcal{D}_R \coloneqq \langle R, \{\phi_r\}_{r \in R} \rangle$, be a pair containing the agent partition and corresponding subspecifications.

A decomposition generates $|R|$ synthesis subproblems that can be solved independently with no coordination between them. Note that $\bigwedge_{r \in R} \phi_r \equiv \phi$. Thus, for $\phi \coloneqq \tau(\psi)$, if $\texttt{Synth}(r, \phi_r)$ returns a valid solution for each subproblem, then the combined solutions satisfy the original specification ψ.

Remark 3. The decomposition method ensures that satisfied subproblems logically entail the original problem. However, we must also guarantee noninterference between signal generators of different teams, i.e., that two subteams do not have opposing goals when servicing predicates. This is problem-dependent: in our experimental results we have one case that avoids this via monotonicity (all goals are to increase signals), one that avoids it via mutual exclusion (never driving the same signal), and one that avoids it with an additional constraint that groups all relevant signals in the same subteam.

4.3 Comparison of Decomposition Candidates

There are many possible decompositions given a team of agents and an STL specification. Our main algorithm requires a method of ranking these candidate decompositions. The algorithm is fully parameterized by the choice of a score, ξ, for each decomposition. Here, we define our choice for this operator.

Since the robustness upper bound condition is necessary but not sufficient, we also introduce the costs c_{ap} and c_{pp} to produce a principled heuristic method for decomposition. These costs consist of a startup cost $c_{ap} : G \times pred(\phi) \rightarrow \mathbb{R}$, capturing the maximum initial cost for an agent of class $g \in G$ to begin servicing a predicate, and a switching cost $c_{pp} : G \times pred(\phi) \times pred(\phi) \rightarrow \mathbb{R}$, capturing the cost for switching from servicing one predicate to another. These costs abstract dynamics or other system properties that influence the feasibility of the synthesis process. The abstraction provides a way of incorporating some dynamics information while remaining computationally efficient (i.e., not solving the full synthesis problem).

For mobile robots, these costs might simply correspond to travel times between regions of the environment. We opt for this more general concept of costs to allow flexibility in the type of problem our framework can be applied to. We assume an analysis procedure that can either exactly or approximately determine these costs given the agent start state, dynamics, and predicate(s). In our experiments, costs are either provided explicitly, or agent states are nodes in a graph, for which we can use standard graph traversal algorithms to determine both startup and switching costs.

Example 4. Let us revisit the computing scenario from Example 2. Since starting from idle has little overhead, c_{ap} can be quite small, representing a few milliseconds to start any arbitrary computing request. However switching from a request that uses many GPU resources to one that requests many CPU resources typically has much higher overhead. Therefore c_{pp} would be higher for switching from GPU-intensive to CPU-intensive tasks or vice-versa.

To design a score, we start by defining the following metrics for a given decomposition:

1. N - number of subteams: $|R|$ (prefer larger)
2. C_{ap} - maximum startup cost: maximum value of c_{ap} over all agents and predicates in the subteam and corresponding subspecification (prefer smaller)
3. C_{pp} - maximum switching cost: maximum value of c_{pp} over all pairs of predicates in a subspecification (prefer smaller)
4. h - maximum horizon: the maximum subspecification horizon (prefer smaller)
5. ρ_{ub} - minimum robustness upper bound: minimum over all subproblem robustness upper bounds; relates predicates to maximum signal value with given decomposition (prefer larger)

Our primary goal is to maximize the number of subteams. More subteams results in smaller synthesis subproblems; however, we include the other metrics (defined in Sec. 2) to discourage "uneven" decompositions that contain subproblems of widely varying difficulty. If one subproblem is nearly as difficult as the original problem, then it still dominates the synthesis time.

Definition 6 (Decomposition Score) *For a given formula ϕ and its decomposition $\mathcal{D}_R$, our* heuristic decomposition score*, $\xi := \langle N, -C_{ap}, -C_{pp}, -h, \rho_{ub} \rangle$ collects the heuristic scores in a tuple.*

Let D_1 and D_2 be two possible decompositions for STL specification, ϕ. For each D_i, let our score choice be $\xi^i \coloneqq \langle N^i, -C^i_{ap}, -C^i_{pp}, -h^i, \rho^i_{ub} \rangle$. Given these metrics, we compare D_1 and D_2 with $\xi^1 >_L \xi^2$, where $>_L$ is a lexicographic comparison. Note that metrics which are preferred to be smaller are negated to prioritize smaller values in the greater-than comparison. We prefer decompositions with a higher score. A decomposition D_i is guaranteed to be infeasible if $\rho^i_{ub} < 0$. Beyond a cheap feasibility check, we included ρ^i_{ub} in our decomposition score to break ties between otherwise equally scored decomposition candidates.

5 Exploring the Formula Rewrite DAG

We now describe an algorithm for decomposing an STL planning task into sub-specifications and agent subteams. We directly explore all possible formula transformations using a Formula Rewrite DAG and choose the formula that gives the best decomposition according to heuristic measures. For efficiency, we also avoid processing nodes of the DAG that do not add a top-level conjunct (and thus cannot increase the number of subteams). Furthermore, we leverage the following theorem to prune nodes that are not worth visiting.

Theorem 4. *Let ϕ be the formula at a node in a Formula Rewrite DAG such that $conj(\phi) = N$. If ϕ has no decomposition into N subteams (one subteam per top-level conjunct) with a nonnegative robustness, then its children do not have decompositions into $N + 1$ subteams. See the Supplementary Materials for a proof sketch.*

Intuitively, Algorithm 1 explores the Formula Rewrite DAG starting from the root and attempting to decompose each transformed formula ϕ into $conj(\phi)$ subteams. It stops the search at nodes with guaranteed infeasible decompositions and compares the decomposition candidates using $>_L$.

Lines 1-2 initialize empty data structures used for tracking nodes to process and infeasible nodes, respectively. Lines 3-4 initialize the candidate to a null value, and the score to the worst possible score. Line 5 pushes the root node of the Formula Rewrite DAG as the start of the search. The loop starting at line 6 processes nodes in the DAG in a breadth-first order until all nodes have been processed or skipped. We assume the queue automatically caches and skips nodes that have already been processed. Lines 7-9 obtain the next node to process and skip it if it is known to be infeasible. Lines 10-13 check if the node has the same number of conjuncts as the parent. If so, it cannot have a greater number of subteams and is skipped. Note that the number of conjuncts only stays the same or increases with formula transformations, so we must still process its children. Line 14 obtains a decomposition assignment and robustness upper bound for the formula. The number of subteams is the number of top-level conjuncts, because we assume each conjunct is assigned a unique subteam. The algorithm still works without this assumption. However, this restriction allows the algorithm to conclude that a formula and all its descendants are infeasible in lines 15-18, by leveraging Theorem 4. If the robustness upper bound is negative,

Algorithm 1 Main decomposition algorithm

```
    Input: Formula Rewrite DAG $G$, Set of agents $\mathcal{A}$
    Output: Decomposition $\mathcal{D}_R$

 1: $Q \leftarrow$ $empty\ queue$                        ▷ Nodes to explore
 2: $D \leftarrow \emptyset$                              ▷ Set to store dropped nodes
 3: $c \leftarrow null$                                   ▷ Start with a null candidate
 4: $\xi \leftarrow worst$                                ▷ Initialize score with worst possible values
 5: $push(Q, G.root)$
 6: while $\neg empty(Q)$ do
 7:     $n \leftarrow pop(Q)$
 8:     if $n \in D$ then
 9:         continue                                      ▷ Formula is known to be infeasible
10:     if $|conj(n)| = |conj(Parent(n))|$ then
11:         $push(Q, children(n))$
12:         $push(D, n)$
13:         continue                                      ▷ Cannot improve on parent
14:     $\mathcal{D}_n, \rho^n_{ub} \leftarrow compute\_assignment(n, \mathcal{A})$
15:     if $\rho^n_{ub} < 0$ then                         ▷ Infeasibility condition
16:         $push(D, Descendants(n))$                     ▷ Descendants are all infeasible
17:         $push(D, n)$
18:         continue
19:     $push(Q, Children(n))$
20:     $\xi^n \leftarrow compute\_score(\mathcal{D}_n, \mathcal{A})$
21:     if $\xi^n >_L \xi$ then                           ▷ lexicographic comparison
22:         $\xi \leftarrow \xi^n$
23:         $c \leftarrow \mathcal{D}_n$
    return $c$
```

all descendants are marked as infeasible to avoid processing them unnecessarily in case they appear on another path of the DAG. Line 19 adds all the node's children onto the end of the queue for future processing and line 20 computes the heuristic score. Lines 21-23 update the best score and candidate decomposition if this is the best score seen thus far according to a lexicographic comparison. Finally, the best decomposition is returned after processing or skipping all nodes in the Formula Rewrite DAG.

The implementation of *compute_score* is specific to the particular heuristic score choice. Depending on the score, there could be additional early-stopping checks before computing the entire decomposition. We efficiently compute the decomposition and score by solving mixed-integer linear programs (MILPs). We provide further information on our MILP encodings for *compute_decomp* and *compute_score* in the Supplementary Material.

Limitations. Our assumption that decomposition assigns a unique subteam to each top-level conjunct allows us to prune descendants. However, it might also rule out solutions that combine top-level conjunctions of a more heavily rewritten formula and achieve a higher number of subteams overall. Although we might miss alternative decompositions, it is important that the decomposition procedure runs quickly. Our goal is to cheaply find a decent decomposition, and let the synthesis procedure proceed from there. Too much upfront computation can be counterproductive for the larger synthesis problem. Empirically, this approach works well.

Despite the computed score, it is still possible that the best decomposition contains an infeasible subproblem that is only revealed during the synthesis procedure. In this case, we can return to the DAG and recover a different decomposition that does not contain the same infeasible subproblem. Future work can investigate weaker transformations that present less risk of creating infeasible problems. Note that between two decomposition options with the same score, the algorithm will pick the one closer to the root node by design of the search procedure. This is desirable because transformations only strengthen the formula, making it harder to satisfy. That is another reason that our restriction to decomposition assignments with one subteam per conjunct is a reasonable heuristic. It tends to stop the search earlier in the DAG even if it produces fewer subteams overall.

Example	Runtime (s)		% Speedup	$\lvert R\rvert$	DAG Size	N Agents	Decomp Time (s)	Largest Subproblem Time(s)
	No Decomp	With Decomp						
Computing	8.05	1.51	81.2	11	5	12	0.59	0.12
Energy Grid	329.37	165.90	49.6	10	128	45	19.74	35.09

Table 1: Results from computing and energy grid case studies. Note that runtime for decomposition is by solving the resulting synthesis problems serially.

6 Experiments and Results

We now provide evidence that this approach surpasses the state-of-the-art, and give two case studies of its application in practical domains. Our implementation is written in Python and encodes MILPs using the `PuLP` Python linear programming toolkit [19]. We instantiate `Synth` with a MILP-based synthesis approach for STL [3]. All results were obtained on a 2.10GHz Intel Xeon Silver 4208 with 64GB of memory. We used `Gurobi` 9.5.1 [12] as the underlying solver for both decomposition and synthesis. Gurobi had access to 16 physical cores with hyperthreading. All comparisons include the time to solve all decomposed subproblems serially. This is an upper bound on the real time, assuming in practice some would be solved in parallel. See the Supplementary Material for more information on our experiments, including an evaluation with the SCIP optimizer 7.0.3 [10] where decomposition has an even larger impact.

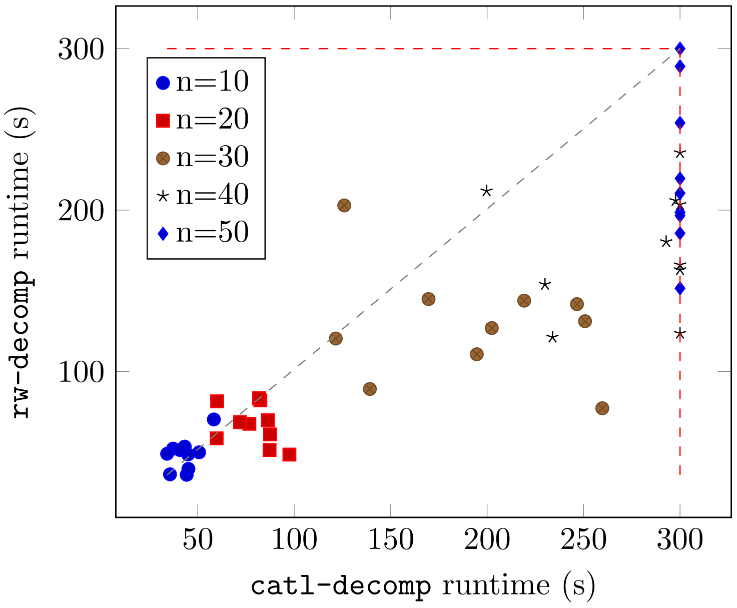


Fig. 3: Comparison of our approach `rw-decomp` to the SMT-based CaTL decomposition approach. Runtime includes the time to decompose and solve each of the subproblems serially.

6.1 CaTL Example

First, we compare against the decomposition technique of [14] for a fragment of STL, Capability Temporal Logic (CaTL). Their approach uses a Satisfiability Modulo Theories (SMT) solver to find agent assignments, and only makes formula transformations after obtaining an assignment. We evaluate our algorithm against theirs on the same set of template formulae used in their paper which vary from 10 to 50 (by tens) randomly-generated agents on a randomly-generated environment. We run 10 trials for each number of agents seeded with the trial number (0-9). We use a timeout of 5 minutes. Fig. 3 depicts our results. Our technique is faster for any value below the diagonal. We define a *degenerate* decomposition as a decomposition with only a single subteam. Our approach had no degenerate solutions and one timeout. The other technique had 2 degenerate solutions and 15 timeouts.

Their technique is faster to decompose than ours (up to 1s vs. up to 13s), but finds less desirable decompositions. This is expected because their technique does not optimize or explore the space of formula rewrites. Both techniques improve the runtime compared to monolithic synthesis.

6.2 Case Studies

We now apply decomposition to two case studies. Table 1 summarizes the results.

Computing Example. We consider a computing cluster whose processor nodes are each equipped with either a CPU or GPU. Each processor has a number of cores available, and can be assigned to process jobs. There is a startup cost associated with sending jobs to processors, and a switching cost for switching from a serial job to parallel and vice versa. The overall formula for a day's requests is

$$\phi_{cluster} = \Diamond_{[0,12)}\phi_{batch} \wedge \Box_{[3,12)}\phi_{admin}, \tag{7}$$

where ϕ_{batch} captures the overnight batch jobs, ϕ_{admin} captures administrative events that must be performed periodically. Both ϕ_{batch} and ϕ_{admin} are conjunctions over sets of individual requests. These requests may specifically ask for a number of CPU cores, a number of GPU cores, or may specify that the job can be accomplished with a CPU and/or GPU. We assume that if a job is running on a node, no other job can start on that node until the previous job has finished. This mitigates the interference concerns in Remark 3.

Energy Grid. We develop an energy grid problem. There are ten towns and a daily specification over half-hour increments. There are two energy companies and each has a power station for each type of energy: coal, natural gas, wind, nuclear, and solar. The specification ensures that each town receives the required power along with additional constraints imposed by each town for cost or green initiatives, such as limiting the amount of coal. Each agent represents 1 GWh from a particular power plant. Since this specification has both less-than and greater-than predicates, we must directly mitigate the interference concerns of Remark 3. We accomplish this with an additional constraint that all signals for a given town must be grouped in the same subteam. This may require bundling several top-level conjuncts into a single conjunct by editing the AST. This prevents the situation in which one subteam is trying to increase a signal while another subteam attempts to decrease it. This specification has a natural geographic decomposition, but the specification needs to be rewritten so that this is possible. Due to the predicate grouping constraint, the original formula can only be decomposed into 3 subteams, but after rewriting we obtain 10 subteams.

7 Conclusions

This work proposed an abstract rewriting system for STL, with proofs of termination and confluence. The rewriting system forms the basis for a method of decomposing an STL specification and distributing it among a heterogenous team of agents. It outperforms a closely related method on a fragment of STL, and is effective on general STL case studies.

Future work includes investigating refinement of formula time bounds in the rewriting system, further formalization of the decomposition procedure, and potential relaxations of the noninterference condition mentioned in Remark 3.

References

1. Baader, F., Nipkow, T.: Term rewriting and all that. Cambridge university press (1999)
2. Banks, C., Wilson, S., Coogan, S., Egerstedt, M.: Multi-agent task allocation using cross-entropy temporal logic optimization. In: 2020 IEEE International Conference on Robotics and Automation (ICRA). pp. 7712–7718. IEEE (2020)
3. Belta, C., Sadraddini, S.: Formal methods for control synthesis: An optimization perspective. Annual Review of Control, Robotics, and Autonomous Systems **2**(1), 115–140 (2019). https://doi.org/10.1146/annurev-control-053018-023717, https://doi.org/10.1146/annurev-control-053018-023717
4. Buyukkocak, A.T., Aksaray, D., Yazıcıoğlu, Y.: Planning of heterogeneous multi-agent systems under signal temporal logic specifications with integral predicates. IEEE Robotics and Automation Letters **6**(2), 1375–1382 (2021)
5. Castanon, D., Drummond, O., Bellovin, M.: Comparison of 2-D assignment algorithms for rectangular, floating point cost matrices. Proc. Os SDI Panels on tracking, No. 4 (1990)
6. Charitidou, M., Dimarogonas, D.V.: Signal temporal logic task decomposition via convex optimization. IEEE Control Systems Letters **6**, 1238–1243 (2021)
7. Chen, Y., Ding, X.C., Stefanescu, A., Belta, C.: Formal approach to the deployment of distributed robotic teams. IEEE Transactions on Robotics **28**(1), 158–171 (2011)
8. Dokhanchi, A., Hoxha, B., Fainekos, G.: On-line monitoring for temporal logic robustness. In: RV. Lecture Notes in Computer Science, vol. 8734, pp. 231–246. Springer (2014)
9. Donzé, A., Maler, O.: Robust satisfaction of temporal logic over real-valued signals. In: International Conference on Formal Modeling and Analysis of Timed Systems. pp. 92–106. Springer (2010)
10. Gamrath, G., Anderson, D., Bestuzheva, K., Chen, W.K., Eifler, L., Gasse, M., Gemander, P., Gleixner, A., Gottwald, L., Halbig, K., Hendel, G., Hojny, C., Koch, T., Le Bodic, P., Maher, S.J., Matter, F., Miltenberger, M., Mühmer, E., Müller, B., Pfetsch, M.E., Schlösser, F., Serrano, F., Shinano, Y., Tawfik, C., Vigerske, S., Wegscheider, F., Weninger, D., Witzig, J.: The SCIP Optimization Suite 7.0. Technical report, Optimization Online (March 2020), http://www.optimization-online.org/DB_HTML/2020/03/7705.html
11. Gundana, D., Kress-Gazit, H.: Event-based signal temporal logic synthesis for single and multi-robot tasks. IEEE Robotics and Automation Letters **6**(2), 3687–3694 (2021)
12. Gurobi Optimization, LLC: Gurobi Optimizer Reference Manual (2022), https://www.gurobi.com
13. Jonker, R., Volgenant, A.: A shortest augmenting path algorithm for dense and sparse linear assignment problems. Computing **38**(4), 325–340 (Dec 1987). https://doi.org/10.1007/BF02278710, https://doi.org/10.1007/BF02278710
14. Leahy, K., Jones, A., Vasile, C.I.: Fast decomposition of temporal logic specifications for heterogeneous teams. IEEE Robotics and Automation Letters (2022)
15. Leahy, K., Serlin, Z., Vasile, C.I., Schoer, A., Jones, A.M., Tron, R., Belta, C.: Scalable and robust algorithms for task-based coordination from high-level specifications (ScRATCHeS). IEEE Transactions on Robotics (2021)
16. Lindemann, L., Dimarogonas, D.V.: Feedback control strategies for multi-agent systems under a fragment of signal temporal logic tasks. Automatica **106**, 284–293 (2019)

17. Liu, S., Saoud, A., Jagtap, P., Dimarogonas, D.V., Zamani, M.: Compositional synthesis of signal temporal logic tasks via assume-guarantee contracts. arXiv preprint arXiv:2203.10041 (2022)
18. Maler, O., Nickovic, D.: Monitoring temporal properties of continuous signals. In: Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems, pp. 152–166. Springer (2004)
19. Mitchell, S., O'Sullivan, M., Dunning, I.: PuLP: A linear programming toolkit for python (2011)
20. Munkres, J.R.: Algorithms for the assignment and transportation problems. Journal of The Society for Industrial and Applied Mathematics **10**, 196–210 (1957)
21. Schillinger, P., Bürger, M., Dimarogonas, D.V.: Decomposition of Finite LTL Specifications for Efficient Multi-agent Planning, pp. 253–267. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-73008-0_18, https://doi.org/10.1007/978-3-319-73008-0_18
22. Sun, D., Chen, J., Mitra, S., Fan, C.: Multi-agent motion planning from signal temporal logic specifications. IEEE Robotics and Automation Letters **7**(2), 3451–3458 (2022)
23. Zhou, Z., Lee, D.J., Yoshinaga, Y., Guo, D., Zhao, Y.: Reactive task allocation and planning of a heterogeneous multi-robot system. arXiv preprint arXiv:2110.08436 (2021)

A Supplementary Material

A.1 Abstract Syntax Tree Formal Definition

Definition 7 (Abstract Syntax Tree) *An* abstract syntax tree *corresponding to an STL formula ϕ is a tuple $\mathcal{T}_\phi = (V, Par, L)$, where V is a set of nodes, $Par : V \to V \cup \{\epsilon\}$ is a parent function, and $L : V \to \Sigma$ is a labeling function.*

The parent relationship indicates which node is the unique parent of a given node. There is exactly one node $v_0 \in V$ with no parent (i.e., $Par(v_0) = \epsilon$), which is the *root* of the AST. Each node is labeled with a symbol in $\sigma \in \Sigma$, the set of operators and predicates appearing in ϕ, i.e., $\Sigma \subseteq pred(\phi) \cup \{\neg, \wedge, \vee, \mathcal{U}_{[a,b)}, \Diamond_{[a,b)}, \Box_{[a,b)}\}$, where $a, b \in \mathbb{R}$ and $a \leq b$.

A.2 AST Examples

For completeness, Fig. 4 shows examples of the ASTs associated with the rewrite DAG in Fig. 2. These are the same ASTs that are seen in Fig. 1b. Fig. 4a is the AST associated with the root formula. Note that there are two top-level conjuncts and thus there is a maximum of two possible sub-formulae. Next, Figs. 4b and 4c show the result of applying $\mapsto_{\mathcal{U}}$ and $\mapsto_{\Box}$, respectively. Both ASTs have three top-level conjuncts, and therefore a maximum of three sub-formulae. Finally, Fig. 4d show the AST for the root formula in decNF. Note that there are four top-level conjuncts, and therefore, this is the maximally decomposable form of the original formula.

A.3 Proof of Theorem 1

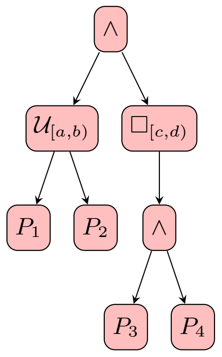
Proof. By Lemma 2.3.3 of [1], a finitely branching rewriting system $(A, \to)$ terminates if there exists a monotone embedding φ from $(A, \to)$ into $(\mathbb{N}, >)$. In our case, φ has two components – the sum of distances of conjuncts and until operators from the root, and the number of occurrences of conjunction and until operators in the formula. Namely,

$$\varphi = \sum_{i=1}^{n_{\wedge,\mathcal{U}}} d_i + (n_{\wedge,\mathcal{U}} + 1) n_{\mathcal{U}} \ , \tag{8}$$

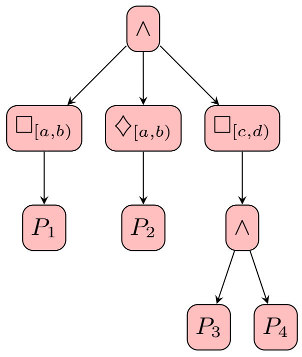
where $n_\wedge$ and $n_{\mathcal{U}}$ are the number of conjunction and until operators appearing in the formula, $n_{\wedge,\mathcal{U}} = n_\wedge + n_{\mathcal{U}}$, and d_i is the distance of the i^{th} conjunction or until from the root in the formula AST.

Both (split-globally) and (split-finally) can only reduce the values of d_i, by moving a conjunction closer to the root. They both leave $n_\wedge$, and $n_{\mathcal{U}}$ unchanged. It remains to prove that (split-until) also leads to a decrease in the value of φ.

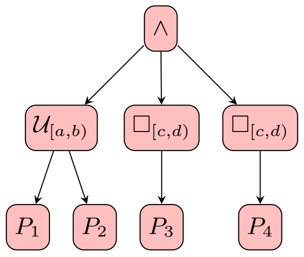
Examining (split-until), we can see two changes occurring in the formula. First, an until is replaced with a conjunction. Second, any downstream operators from that until have their distance to the root increased by one, because of



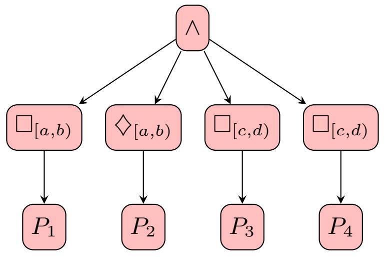
(a) Abstract syntax tree for root in Fig. 2



(b) Abstract syntax tree for node after applying $\mapsto_{\mathcal{U}}$ in Fig. 2



(c) Abstract syntax tree for node after applying $\mapsto_{\square}$ in Fig. 2



(d) Abstract syntax tree for node after applying $\mapsto_{\mathcal{U}}$ and $\mapsto_{\square}$ in Fig. 2

Fig. 4: Details of abstract syntax trees seen in Fig. 1, created from the rewrite DAG in Fig. 2.

the insertion of a globally or eventually operator. Put formally, the (split-until) operation results in the following changes:

$$n_{\mathcal{U}} \to n_{\mathcal{U}} - 1 \tag{9}$$

$$n_{\wedge} \to n_{\wedge} + 1 \tag{10}$$

$$n_{\wedge,\mathcal{U}} \to n_{\wedge,\mathcal{U}} \tag{11}$$

$$d_i \to d_i + 1\,, \tag{12}$$

where (12) is the worst case change in d_i. For any conjunctions or until operators downstream from the (split-until), the distance d_i is increased by one due to the insertion of temporal operators into the formula. All other d_i are unchanged. Substituting this information into (8), we obtain

$$\varphi' = \sum_{i=1}^{n_{\wedge,\mathcal{U}}} (d_i + 1) + (n_{\wedge,\mathcal{U}} + 1)(n_{\mathcal{U}} - 1)\,. \tag{13}$$

Expanding and gathering terms yields the following

$$\begin{aligned}
\varphi' &= \sum_{i=1}^{n_{\wedge,\mathcal{U}}} d_i + \sum_{i=1}^{n_{\wedge,\mathcal{U}}} 1 + (n_{\wedge,\mathcal{U}} + 1)n_{\mathcal{U}} - (n_{\wedge,\mathcal{U}} + 1) && (14)\\
&= \sum_{i=1}^{n_{\wedge,\mathcal{U}}} d_i + n_{\wedge,\mathcal{U}} + (n_{\wedge,\mathcal{U}} + 1)n_{\mathcal{U}} - n_{\wedge,\mathcal{U}} - 1 && (15)\\
&= \varphi - 1\,. && (16)
\end{aligned}$$

Therefore, for all replacement rules we consider, φ is a monotone embedding into $(\mathbb{N}, >)$, and our rewriting system terminates. □

A.4 Proof of Theorem 2

To prove Theorem 2, we first provide the following definitions.

Definition 8 (Commutativity) *Two reduction systems, $\to_1$ and $\to_2$ are said to* commute *if* $\forall x\ .\ y_1 \overset{*}{\leftarrow}_1 x \overset{*}{\to}_2 y_2 \implies \exists z\ .\ y_1 \overset{*}{\to}_2 z \overset{*}{\leftarrow}_1 y_2$.

For a given reduction system $\to$, let $\overset{=}{\to}$ denote its reflexive closure. The reduction systems *strongly* commute if $\forall x\ .\ y_1 \leftarrow_1 x \to_2 y_2 \implies \exists z\ .\ y_1 \overset{=}{\to}_2 z \overset{*}{\leftarrow}_1 y_2$. A special case of strongly commuting is the *commuting diamond property*, $\forall x\ .\ y_1 \leftarrow_1 x \to_2 y_2 \implies \exists z\ .\ y_1 \to_2 z \leftarrow_1 y_2$. This property shows that a single step of each reduction creates a commuting diagram (Fig. 5).

Proof. To prove confluence, we rely on two lemmas from [1]. Lemma 2.7.10 states that the union of two reduction systems that are confluent and commute, is also confluent. Lemma 2.7.11 states that two reduction systems commute if they strongly commute.

We break our formula rewriting reduction system into three independent reduction systems:

$$\begin{tikzcd} x \arrow[r, "2"] \arrow[d, "1"'] & y_2 \arrow[d, "1"] \\ y_1 \arrow[r, "2"] & z \end{tikzcd}$$

Fig. 5: Illustration of the commuting diamond property.

1. $\rightarrow_{\Box} := \{\mapsto_{\Box}\}$: reduction system for (split-globally)
2. $\rightarrow_{\Diamond} := \{\mapsto_{\Diamond}\}$: reduction system for (split-finally)
3. $\rightarrow_{\mathcal{U}} := \{\mapsto_{\mathcal{U}}\}$: reduction system for (split-until)

Our full formula rewriting reduction system is $\rightarrow_{STL} := \rightarrow_{\Box} \cup \rightarrow_{\Diamond} \cup \rightarrow_{\mathcal{U}}$. We will prove confluence by proving these reduction systems confluent and commutative, then building up to our full reduction system. Each individual reduction system is trivially confluent.

$\rightarrow_{\Box}$ and $\rightarrow_{\Diamond}$ have the commuting diamond property: For arbitrary ϕ, we assume the precondition of the commuting diamond property, $\phi_1 \leftarrow_{\Box} \phi \rightarrow_{\Diamond} \phi_2$. The two reduction systems make a local change to the AST. In particular, they map $\circ_{[a,b]}(\psi_1 \wedge \psi_2) \mapsto (\Box_{[a,b)}\psi_1 \wedge \Box_{[a,b)}\psi_2)$, for $\circ \in \{\Box_{[a,b)}, \Diamond_{[a,b)}\}$. Note that subformulae ψ_1 and ψ_2 are not changed. Since $\rightarrow_{\Box}$ and $\rightarrow_{\Diamond}$ operate on different temporal operators, they cannot be applied to the same node in the AST. Furthermore, they must be applied to a temporal operator over a conjunction. Therefore it is not possible that one rewrite is applied to a direct child of the subterm of ϕ rewritten by the other. Thus, each can be applied independently, making local changes to non-overlapping regions of the AST, and the final ASTs are the same.

The commuting diamond property implies strong commutativity, which implies commutativity. Since, $\rightarrow_{\Box}$ and $\rightarrow_{\Diamond}$ are confluent and commute, $\rightarrow_{\Box\Diamond} := \rightarrow_{\Box} \cup \rightarrow_{\Diamond}$ is also confluent.

$\rightarrow_{\Box\Diamond}$ and $\rightarrow_{\mathcal{U}}$ have the commuting diamond property: Following the same reasoning as above, these two operations must operate on different nodes. Similarly, for rewrites impacting non-overlapping regions of the AST, the final ASTs are the same. Unlike above, rewrites can be applied to direct children for nodes of the form $\circ_{[a,b]}(\psi_1 \wedge \psi_2)\mathcal{U}_{[c,d)}\psi$, or $\psi\mathcal{U}_{[c,d)} \circ_{[a,b]} (\psi_1 \wedge \psi_2)$. We handle each of these cases independently.

Case 1: $\circ_{[a,b]}(\psi_1 \wedge \psi_2)\mathcal{U}_{[c,d)}\psi$ is reduced to $\Box_{[0,c)}(\Box_{[a,b)}\psi_1 \wedge \Box_{[a,b)}\psi_2) \wedge \Diamond_{[c,d)}\psi$ for both $\rightarrow_{\Box\Diamond}$ followed by $\rightarrow_{\mathcal{U}}$ and the reverse.

Case 2: $\psi\mathcal{U}_{[c,d)} \circ_{[a,b]} (\psi_1 \wedge \psi_2)$ is reduced to $\Box_{[0,c)}\psi \wedge \Diamond_{[c,d)}(\Box_{[a,b)}\psi_1 \wedge \Box_{[a,b)}\psi_2)$ for both $\rightarrow_{\Box\Diamond}$ followed by $\rightarrow_{\mathcal{U}}$ and the reverse.

This covers all the cases and proves the commuting diamond property. Since $\rightarrow_{\Box\Diamond}$ and $\rightarrow_{\mathcal{U}}$ are both confluent and commute, then $\rightarrow := \rightarrow_{\Box\Diamond} \cup \rightarrow_{\mathcal{U}} = \rightarrow_{\Box} \cup \rightarrow_{\Diamond} \cup \rightarrow_{\mathcal{U}}$ is also confluent. $\Box$

A.5 Proof Sketch for Theorem 4

Proof (Sketch). Let there be an edge from ϕ to ϕ' in the rewrite DAG, $conj(\phi) = N$, and assume ϕ has no decomposition with N subteams and a nonnegative ρ_{ub}.

Case 1. $conj(\phi') = conj(\phi)$ The formula transformation did not introduce a new top-level conjunct. Trivially there cannot be $N + 1$ subteams for ϕ'.

Case 2. $conj(\phi') = conj(\phi)+1$ The formula transformation introduced a new top-level conjunct. Let $\phi \coloneqq \psi_1 \wedge \cdots \wedge \psi_k$ for $k \geq 1$. Without loss of generality, we assume ψ_k was rewritten by $\psi_k \mapsto \psi_{k1} \wedge \psi_{k2}$. Note that all agent assignments to ϕ have negative robustness upper bound. Thus, we consider an arbitrary assignment via universal instantiation. By the definition of robustness upper bound in (5), at least one of the top-level conjuncts had a negative robustness upper bound under this assignment. If ψ_k did not have a negative robustness, then one of the other conjuncts was the limiting factor and the rewrite will not change this. If ψ_k did have negative robustness upper bound, then we must show that its robustness cannot be improved. Note by (5) that robustness upper bound is dependent on the assigned agents. For any of the assignments, we cannot increase the robustness upper bound of $\psi_k \coloneqq \psi_{k1} \wedge \psi_{k2}$ by splitting agents between ψ_{k1} and ψ_{k2} because there will be less assigned agents to each.

A.6 Decomposition Implementation Details

We now describe the individual components of our decomposition algorithm in more detail. We start with *compute_assignment*. Our implementation splits this procedure into two steps: class assignment, and individual agent assignment.

Class Assignment We start by describing a general class assignment algorithm, then discuss a restriction which emits useful bounds for pruning the Formula Rewrite DAG as described in Theorem 4. The class assignment encoding computes a decomposition and assignment of number of agents from a given signal class to each subteam. It assumes that the maximum possible decomposition is to assign each conjunct to its own subteam. Since this may not be possible, the encoding must allow for grouping two or more conjuncts in the same subteam. The class assignment problem is encoded as an ILP.

Let M be the number of top-level conjuncts. We define the following variables:

1. I: a $M \times M$ matrix of indicator variables, where I_{ij} is 1 *iff* conjunct i is assigned to subteam j;
2. z_j^g for $0 \leq j < M$ and $g \in G$: an integer between 0 and $|\mathcal{A}|$ encoding the number of agents of class g assigned to subteam j. We refer to all these variables collectively as z;
3. ρ_k for $0 \leq k \leq ast(\phi)$: the robustness upper bound for each node in the formula AST.
4. N: the number of subteams

$$\begin{aligned} max\ & \rho_{max} N + \rho_{root} \\ s.t.\ & N = \Sigma_j^M max_i(I_{ij}) \\ & \forall j\ \Sigma_i I_{ij} = 1 \\ & \forall g \in G\ \Sigma_j z_j^g = |\{a \in \mathcal{A} | class(a) = g\}| \\ & \forall k \in \mathcal{T}_\phi\ \rho_k = compute\_robustness(\phi, k, I, z) \\ & \rho_{root} \geq 0 \end{aligned} \tag{17}$$

The ILP encoding given in (17) maximizes the number of subteams and the robustness upper bound for this assignment at the root. The number of subteams is multiplied by a (possibly loose) upper bound of ρ_{root} given the available agents, ρ_{max}, to ensure that increasing the number of subteams has a larger impact on the objective than increasing robustness upper bound. We prefer the maximum number of subteams possible as long as the robustness upper bound at the root is nonnegative. The first constraint encodes the number of subteams using the indicator variables. The second constraint requires that each conjunct is assigned to exactly one subteam. The third constraint requires the sum of all subteam class assignments to equal the expected number of agents of that class. The fourth constraint encodes robustness upper bound for each AST node according to the recursive definition. The number of agents of each capability class at a given AST node is a straightforward function of I, z, and the AST structure (children inherit the assignment of parent nodes). The fifth constraint requires that the robustness upper bound be nonnegative.

In practice, we use a restricted version of (17) that is more performant. The restriction assumes that each subteam corresponds to a single top-level conjunct and simply aims to maximize the robustness upper bound. Taking this into account results in a much smaller ILP encoding, and enables us to prune descendants in the DAG by utilizing Theorem 4. Instead of optimizing over the number of subteams and the robustness upper bound at the root, we fix the number of subteams and only search for an assignment that maximizes the robustness upper bound. If the optimal value is negative, then the problem is infeasible. This scales much better and is a reasonable approach in many situations when paired with the rewriting framework that adds top-level conjuncts. It will simply stop exploring a node's children when there are more conjuncts than possible subteams. If the root node already has more top-level conjuncts than possible subteams, then we can run the class assignment algorithm given in (17) to obtain a class assignment.

Individual Agent Assignment For each signal class, we solve a separate MILP to assign individual agents. Each agent has the same set of signals but can start in separate states. Let N^* be the optimal number of subteams returned by the class assignment and A_c be the set of agents for class c. With some abuse of notation we use $A_c(i)$ to refer to the agents assigned to subteam i. We start by defining the following variables:

1. C: a $N^* \times |A_c|$ matrix of constant start up costs for each agent and subspecification (maximum agent to predicate cost over all predicates in the subspecification) as referred to in Sec. 4.3
2. B: a $N^* \times |A_c|$ matrix of indicator variables for whether an agent is matched to a particular subteam and subspecification

The MILP encoding is given by:

$$\begin{aligned} min\ & \Sigma_i \Sigma_j C_{ij} B_{ij} \\ s.t.\ & \Sigma_j B_{ij} = |A_c(i)| \ \ \forall i \\ & \Sigma_i B_{ij} = 1 \ \ \forall j \end{aligned} \tag{18}$$

It minimizes the total assignment cost while ensuring the assignment matches the class assignment A_c and assigns every agent to a single subspecification. We solve this MILP for each signal class. Note that we cannot use a polynomial-time assignment algorithm such as Munkres [20] or JVC [13,5] because each subspecification can have multiple agents assigned to it.

Compute Score Finally, we describe our implementation of *compute_score* for the heuristic metrics described in Def. 6. Recall that our overall score was given by: $\xi \coloneqq \langle N, -C_{ap}, -C_{pp}, -h, \rho_{ub} \rangle$.

We set $N \coloneqq N^*$ and $\rho_{ub} \coloneqq \rho^*_{root}$, the optimal values from the class assignment optimization. The start up cost C_{ap} is the maximum over all objective values from individual agent assignment. The predicate to predicate cost C_{pp} is problem-specific. In all of our experiments and case studies, we had a graph representation of the environment labelled with predicate symbols or explicitly provided costs. Individual predicate-to-predicate costs (represented by the function c_{pp}) were calculated as maximum weighted shortest path between any two states labelled with the given predicates in the graph representation. The predicate-to-predicate cost for a given subspecification was the maximum over the pairwise costs between all predicates in a subspecification. Then C_{pp} is defined as the maximum over all the subspecification predicate-to-predicate costs. The formula horizon h is calculated as defined in Def. 4.

A.7 Case Study Details

Here we provide implementation details related to our specific case studies.

CaTL Example When comparing our method against the decomposition method in [14], we perform a two-tailed Wilcoxon Signed Rank Test on the runtime difference for each fixed number of agents (10-50) and obtain test statistics: 10, 10, 6, 1, and 1, respectively. These are all statistically significant for $\alpha = 10\%$ and the last two provide confidence up to $\alpha = 0.5\%$. Timeouts were included with the timeout value of 5 minutes. Note that the first test statistic (for 10 agents) is statistically significant, but in favor of the other technique.

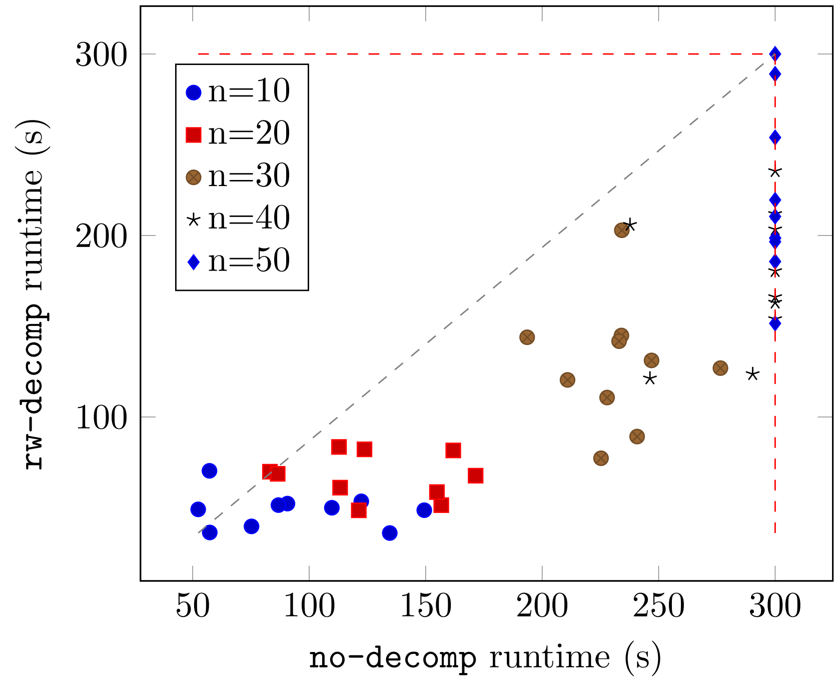


Fig. 6: Comparison of our approach `rw-decomp` to monolithic synthesis. The decomposition runtime includes the time to decompose and solve each of the subproblems serially.

Figure 6 depicts the runtimes of our approach against monolithic (no decomposition) synthesis. The monolithic approach had 17 total timeouts and our approach had one. This demonstrates that decomposition has value over solving the entire problem centrally. The approach of [14] also outperformed monolithic synthesis.

We also ran these same experiments with SCIP 7.0.3 [10]. Figures 7 and 8 show the results compared to the SMT-based CaTL decomposition approach and monolithic synthesis, respectively. Monolithic synthesis resulted in 31 timeouts. The SMT-based approach had two degenerate solutions and 22 timeouts. Our approach had no degenerate solutions and five timeouts.

Compute Example In this example, each agent is a compute node with either a CPU (C) or GPU (G), equipped with a certain number of cores that can be process a job in serial (S) or in parallel (P). There are two types of jobs that may get submitted for processing – batch jobs (B) or administrative jobs (A), and each has its own unique request number. Batch jobs get executed once in a 12 hour period, while administrative jobs are executed periodically. Every job requires a certain amount of compute capability for a certain amount of time.

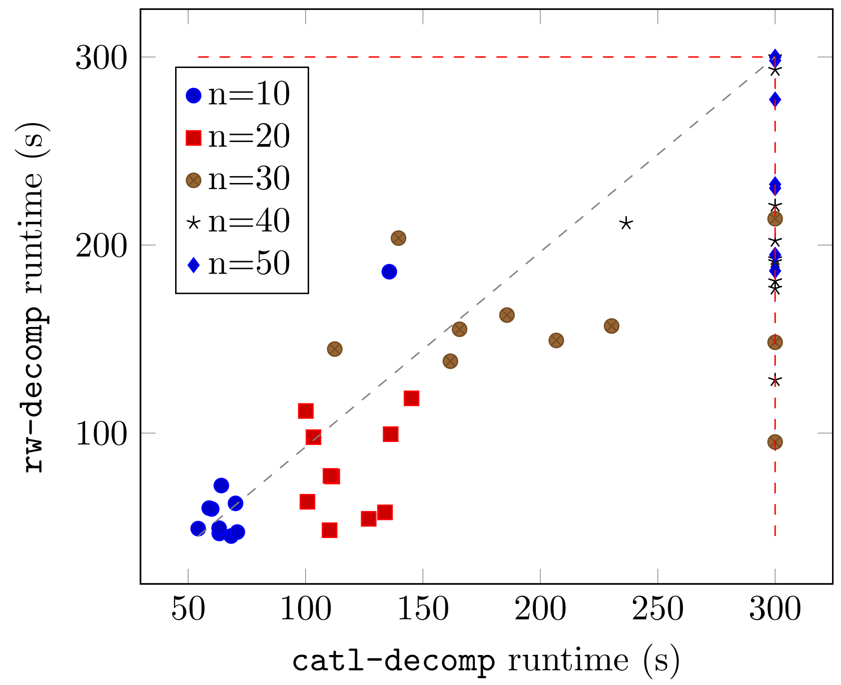


Fig. 7: Comparison of our approach `rw-decomp` to the SMT-based CaTL decomposition approach using SCIP.

For example $\Box_{[0,2)}(A_3 C_P \geq 16)$ is an administrative job with request number 3, and it requires at least 16 CPU cores running in parallel for 2 time steps. A scheduler must assign each job to one or more compute nodes so that the job can run for the specified period of time. A new job cannot start on a given node

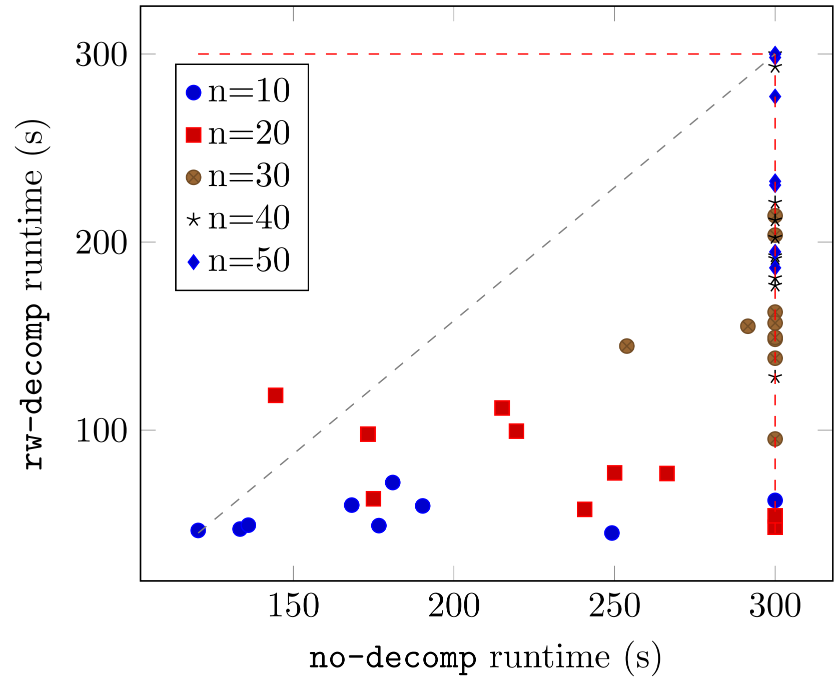


Fig. 8: Comparison of our approach `rw-decomp` to monolithic synthesis using SCIP.

until its previous job has completed. The full specification for this problem is

$$\begin{aligned}
&\Diamond_{[0,12)}\big(\Box_{[0,4)}(B_0C_P \geq 16)\lor \\
&\qquad \Box_{[0,2)}(B_0G_P \geq 10)\big)\land \\
&\Diamond_{[0,12)}\big(\Box_{[0,2)}(B_1C_P \geq 20)\lor \\
&\qquad \Box_{[0,1)}(B_1G_P \geq 10)\big)\land \\
&\Diamond_{[0,12)}\big(\Box_{[0,4)}(B_2C_S \geq 8)\big)\land \\
&\Diamond_{[0,12)}\big(\Box_{[0,1)}(B_2G_P \geq 8)\lor \\
&\qquad \Box_{[0,2)}(B_2G_S \geq 16)\big)\land \\
&\Diamond_{[0,12)}\big(\Box_{[0,4)}(B_4C_S \geq 8)\big)\land \\
&\Box_{[3,12)}\Big(\Diamond_{[0,3)}\big(\Box_{[0,1)}(A_0G_S \geq 8)\lor \\
&\qquad \Box_{[0,1)}(A_0C_S \geq 8)\big)\land \\
&\qquad \Diamond_{[0,3)}\big(\Box_{[0,1)}(A_1C_S \geq 8)\big)\land \\
&\qquad \Diamond_{[0,6)}\big(\Box_{[0,1)}(A_2G_P \geq 8)\lor \\
&\qquad \Box_{[0,2)}(A_2C_P \geq 16)\big)\land \\
&\qquad \Diamond_{[0,6)}\big(\Box_{[0,1)}(A_3G_P \geq 8)\lor \\
&\qquad \Box_{[0,2)}(A_3C_P \geq 16)\big)\land \\
&\qquad \Diamond_{[0,6)}\big(\Box_{[0,1)}(A_4G_P \geq 8)\land \\
&\qquad \Box_{[0,1)}(A_4C_S \geq 16)\big)\Big)\,.
\end{aligned} \tag{19}$$

We also ran this problem using SCIP. Without decomposition it took 23.78s to solve. The decomposition approach took 3.43 to decompose and solve serially, where computing the decomposition took 1.3s and the longest subproblem took 0.28s to solve.

Energy Grid We have power companies A and B. Each agent in the energy example contains a signal for referring to overall power P and of a particular type C=coal, NG=natural gas, W=wind, N=nuclear, and S=solar for each region i. Thus, each power station can supply power to any region(s). Overall power and energy type are always applied together. This allows us to specify overall power requirements while also making statements about the power sources. The naming scheme for signals used in specifications is $\langle\texttt{type}\rangle\langle\texttt{company}\rangle\langle\texttt{region}\rangle$. For example, CA_0, refers to coal from power company A applied to region 0. The region could request generic power with P_0, which does not specify the source. To avoid conflicting signal generators across subteams (one less than predicate trying to reduce power, and another trying increase power in different subteams), we add an additional pass that groups conjuncts such that regions are not split across more than one subteam. Note that each subteam can still contain multiple regions. The full energy-grid specification is given by

$$\begin{aligned}
&\Diamond_{[5,10)}(\Box_{[0,43)}(P_0 \geq 9 \wedge P_1 \geq 4)\wedge \\
&\qquad\Box_{[0,43)}(P_2 \geq 7 \wedge P_3 \geq 10)\wedge \\
&\qquad\Box_{[0,43)}(P_4 \geq 3 \wedge P_5 \geq 3)\wedge \\
&\qquad\Box_{[0,43)}(P_6 \geq 3 \wedge P_7 \geq 2))\wedge \\
&\Box_{[5,46)}(\Diamond_{[0,1)}(SA_0 \geq 2 \vee NA_0 \geq 3)\wedge \\
&\qquad\Diamond_{[0,2)}(SA_1 \geq 2 \vee NA_1 \geq 4)\wedge \\
&\qquad\Diamond_{[0,3)}(NGA_2 \geq 2 \vee WA_2 \geq 1)\wedge \\
&\qquad\Diamond_{[0,4)}(NGA_3 \geq 2 \vee WA_3 \geq 2))\wedge \\
&\Diamond_{[5,10)}(\Box_{[2,43)}(CA_1 \leq 2))\wedge \\
&\Box_{[5,46)}(\Diamond_{[0,1)}(SB_4 \geq 2 \vee NB_4 \geq 3)\wedge \\
&\qquad\Diamond_{[0,2)}(SB_5 \geq 2 \vee NB_5 \geq 4)\wedge \\
&\qquad\Diamond_{[0,3)}(NGB_6 \geq 2 \vee WB_6 \geq 1)\wedge \\
&\qquad\Diamond_{[0,4)}(NGB_7 \geq 2 \vee WB_7 \geq 2))\wedge \\
&\Diamond_{[5,10)}(\Box_{[2,43)}(CB_4 \leq 2))\wedge \\
&\Diamond_{[5,10)}(\Box_{[2,43)}(SB_8 \geq 1))\wedge \\
&\Diamond_{[5,10)}(\Box_{[2,43)}(SB_9 \geq 1 \vee WB_9 \geq 2))\wedge
\end{aligned} \tag{20}$$

We ran this case study using SCIP as well. Without decomposition, this problem timed out at 6 hours. With decomposition it took 189.72s to decompose and solve the subproblems serially, where computing the decomposition took 23.67s and the longest subproblem took 30.96s to solve.