Episodic Constraint Satisfaction Problems: Leveraging Temporal Constraints for Efficient Decision Making under Durative Constraints

No Author Given

No Institute Given

Abstract. For real-world applications such as network configuration, robotics, and space systems, automated systems will need to make decisions on what actions to take and when, given constraints governing state evolution, and constraints describing user specified outcomes. Constraint programming (CP) is a natural approach for decision making in such systems.
While efficient specialized algorithms have been developed to perform scheduling under timing constraints, there have been few extensions to allow such methods to handle complex state constraints over continuous time. On the other hand, while recent advances in CP have allowed scalable solutions to difficult problems with novel constraint specifications, such approaches do not take advantage of the structure inherent in cases where timing constraints are a significant part of the problem.
In this paper, we propose the Episodic Constraint Satisfaction Problem (ECSP), a constraint programming paradigm which allows explicit representation of temporal constraints. We present an solver that exploits temporal constraints to efficiently decompose and solve ECSPs with mild restrictions on the class of constraints, while maintaining execution flexibility. This solver is comprised of three interacting layers: the ordering layer, the temporal layer and the state constraint layer, accelerating the search process by leveraging ordering conflicts across the layers. We empirically show the advantage of this approach in a set of benchmark problems, and demonstrate a 100 times speed up against state-of-the-art solvers working on equivalent numerical programming and CP encodings.

1 Introduction

A wide range of real-world problems feature decision making given tightly coupled time and state constraints. An automated system commanding multiple underwater vehicles to explore the seafloor [15] must decide on control signals for vehicles given the presence of currents, and timing and spatial requirements on data-gathering activities. A mission aware network configuration manager [3] must plan routes and forward error correction settings for flows with requirements on bandwidth, loss, delay, and time of completion. In both instances, the decision maker must decide on not only a course of action, but also the time at which each action is executed. These decisions must also conform to constraints which describe requirements on timing, the dynamics of the system, and goal specifications. There is thus a strong demand for automated systems which schedule tasks and configure system settings under various complex state constraints with critical time requirements.

Such automated planning problems are naturally represented with the constraint programming (CP) paradigm, in which a general class of constraints are specified over a set of decision variables to be assigned by the program. Recent advances in CP techniques have lead to remarkable improvements in scalability for classical operations research (OR) domains, including Vehicle Routing Problem with Time Windows (VRPTWs) and Job Shop Scheduling Problems (JSSPs) [9]. Although researchers have found success in applying CP techniques to solve scheduling problems with complex resource constraints such as cumulative [12], there has been less focus on modeling over continuous time. While standard CP formulations are expressive enough to capture coupled problems with timing and state constraints, they are unable to leverage recent works in efficient checking algorithms for temporal constraints [14] or adaptive execution [11].

One particularly successful line of research in scheduling given temporal constraints is based on the Simple Temporal Network (STN) formalism [4], which models the temporal constraints as feasible durations between event. The complexity of consistency checking for STNs has been proven to be polynomial. Temporal Planning Networks (TPNs) extend STNs with state constraints [8], and play an important role in adaptive execution [10]. Defining a set of events and activities, Qualitative State Plans (QSPs) are similar to TPNs, and successfully applied in hybrid planning especially in bipedal walking problems [7]. However, existing works on TPNs and QSPs only consider basic logic or linear constraints, and do not have the expressiveness enabled by more complex constraints. This motivates an alternative formalism for specifying constraint programs with complex coupled state over continuous durations.

The key insight of this paper is that a CP formalism which allows explicit declaration of temporal constraints is useful in three ways. First, it allows more natural modelling of problems. More importantly, it allows a solver to exploit temporal constraints in solving the CP: the temporal constraints are not only handled by efficient subsolvers from the STN literature, but also leveraged to decompose each problem instance into independent subproblems for a divide-and-conquer solution approach. In addition, by using temporal constraints in STNs, the output schedule is flexible and adaptive.

In this paper, we propose the Episodic Constraint Satisfaction Problem (ECSP), a new class of constraint satisfaction problems, to model temporal problems with complex state constraints. ECSPs extend the notion of variables and constraints to state variables and episodes, and the elements of domains are extended to arbitrary state trajectories that also capture the temporal aspect of feasible assignments over continuous time. For a class of ECSP known as Time-insensitive ECSPs, in which constraints are not time varying within an episode, we construct a method for decomposing ECSPs into independent subproblems by analyzing the temporal constraints. Again leveraging temporal constraints in the specifications, we further develop an ordering-based ECSP solver to solve Time-insensitive ECSPs. In our algorithms, we extend the notion of Minimal Critical Sets (MCSs) [2] to Main Components (MCs), and decompose our ordered instances into several MCs. To accelerate our solver, we also provide two conflict mechanisms that reason over MCs.

2 Problem Formulation

In this section, we define a new problem formalism for constraint programs with coupled state and temporal constraints which allows explicit declaration of temporal constraints and durative state constraints. We begin with a pedagogical example, which is trivial but intricate-structured to help explain the basic definitions and the solving approach in the rest of the paper.

In Figure 1, we consider a task scheduling problem formulated as an ECSP that will be introduced in the next section. In this example, a machine is going to process two tasks and the scheduler needs to provide feasible schedules for these tasks. However, this machine requires 120 seconds to warm up, and cannot process any task before being ready.

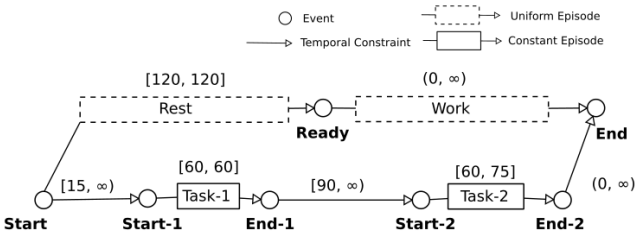


Fig. 1: A task scheduling example: a circle represents an event which is a point in time. An arrow labeled with an interval between a pair of events is a temporal constraint that specifies the feasible duration of the event pair, and a rectangle with an edge on the left and an arrow on the right represents an episode that specifies a requirement between the event pair. While solid-line rectangles are constant episodes where each scoped task cannot flip between being active and inactive, dotted rectangles are uniform episodes.

2.1 Episodic Constraint Satisfaction Problems

In the section, we introduce the formal definitions of Episodic Constraint Satisfactions Problems (ECSPs), which extend CSPs with the notion of time to represent scheduling problems with complex state constraints over continuous time. We also classify the episodes according to their temporal and state effects. Then, we discuss a tractable subset of ECSPs called Time-insensitive ECSPs, and demonstrate its modeling efficiency by representing an equivalent CSP formulation.

Definition 1. *(Event) An event is a real variable or an integer variable.*

Definition 2. *(State Variable) A state variable maps time $t \in \mathbb{R}_{\geq 0}$ to the values from an associated domain.*

Definition 3. *(State Trajectory) A state trajectory of S over I maps $t \in I$ to an assignment of S, where S is a vector of state variables and I is a continuous interval. S and I are said to be the scope and the interval of this state trajectory.*

Definition 4. *(Episode) An episode is a a tuple* $\langle e^{\vdash}, e^{\dashv}, D, S, SC \rangle$ *, where:*

- $e^{\vdash}$ *and* $e^{\dashv}$ *are the start and end events of the episode.*
- D *is the duration constraint, and is given by a tuple* $\langle D_l, D_u \rangle$ *such that* $D_l \leq e^{\dashv} - e^{\vdash} \leq D_u$*.*
- S *is the scope, and is given by a set of state variables.*
- SC *is the state constraint, and is given by a Boolean function of state trajectories scoping on* S*.*

Definition 5. *(ECSP) An Episodic Constraint Satisfaction Problem is a tuple* $\langle Ev, SV, Ep \rangle$*.* Ev *is the set of events.* SV *is the set of state variables.* Ep *is the set of Episodes.*

Definition 6. *(ECSP Solution) An ECSP solution is a tuple* $\langle s, st \rangle$*, where:*

- s *is a schedule that assigns every event with a value, such that all the duration constraints are satisfied.*
- st *is a state trajectory of all the state variables from the earliest time to the latest time of* s*, such that all the state constraints are satisfied.*

2.2 Time-insensitive ECSPs

In this section, we consider a special class of episodes with additional restrictions on the set of episodes. This is of interest because ECSPs featuring only these types of episodes are tractable as shown in Section 3.

Before we proceed, we note that an episode that constrains only the duration — i.e., its state constraint is trivially *true* for every state trajectory satisfying the duration constraint — is equivalent to a simple temporal constraint [4]. As a result, a temporal constraint can be extracted from an episode by modifying its state constraint to be trivially *true*.

Definition 7. *(Uniform and Constant Episode) A uniform episode applies the same constraint to the scoped state variables regardless of the time interval. A* constant *episode is a uniform episode*[1] *where the assignments to the scoped state variables must be constant.*

For instance, that a car can carry no more than 4 passengers during a trip is described by an episode with the state constraint $\langle uniform, 0 \leq numP \leq 4 \rangle$. If the number of passengers cannot be changed, the state constraint will be $\langle constant, 0 \leq numP \leq 4 \rangle$.

Definition 8. *(Time-insensitive Episode) A uniform episode is time-insensitive if and only if the constraint applied on the scoped state variables is not changed when the time interval varies.*

[1] In the remainder of this paper, we use "uniform episode" to refer to uniform but non-constant episodes.

The state constraint SC of a time-insensitive episode can be interpreted as a tuple $\langle uniform, C\rangle$ or $\langle constant, C\rangle$, where C is a constraint as defined in classical CSPs. Both episodes in the previous passenger examples are time-insensitive. However, if the driver is unwilling to drive alone when a trip is longer than 30 minutes, the state constraint $\langle uniform, (0 \leq numP \leq 4) \wedge ((Ev^{\dashv} - Ev^{\vdash} > 30) \rightarrow (numP > 0))\rangle$ will lead to a time-sensitive episode. Another time-sensitive example is an episode with continuous effects such as decreasing batteries.

Time-insensitive ECSPs are a tractable subset of ECSPs capable of modeling a large portion of real world problems such as multiple autonomous underwater vehicles to explore the seafloor and temporal configuration problems. Although they are not strictly more expressive than general CP or numerical programming encodings, Time-insensitve ECSPs are more concise and allow us to leverage results from the STN literature.

The running example in Figure 1 can be modelled as a Time-insensitive ECSP with seven events and one state variable — an integer c where a positive number means that a task indexed by this positive number is running, 0 means the machine is idle, and negative numbers represent warm-up phases. We have the following uniform episodes:

$$Rest = \langle Start, Ready, \langle 120, 120\rangle, \langle c\rangle, \langle uniform, c < 0\ \rangle\rangle$$
$$Work = \langle Ready, End, \langle 0, \infty\rangle, \langle c\rangle, \langle uniform, c \geq 0\ \rangle\rangle$$

The constant episodes are as follows:

$$Task\text{-}1 = \langle Start\text{-}1, End\text{-}1, \langle 60, 60\rangle, \langle c\rangle, \langle constant, c = 1\ \rangle\rangle$$
$$Task\text{-}2 = \langle Start\text{-}2, End\text{-}2, \langle 60, 75\rangle, \langle c\rangle, \langle constant, c = 2\ \rangle\rangle$$

We also have three simple temporal constraints. For example, the constraint between $Start$ and $Start$-1 is $\langle Start, Start\text{-}1, \langle 15, \infty\rangle, \langle\ \rangle, *\rangle$.

This scheduling example can also be modeled as a simple temporal problem. Note that, ECSPs are defined to model problems with complex state constraints over time, while this scheduling example is used for the convenience of presenting the Time-insensitive ECSP-Solver in section 3.

2.3 CP Formulation

As an example, we walk through encoding a Time-insensitive ECSP with n events and m episodes in a CP formulation.

We begin with extracting events and temporal constraints. Each event will be represented by a variable with its original domain, and we collect the Boolean constraint $D_l \leq e^{\vdash} - e^{\dashv} \leq D_u$ from each episode.

To represent state constraints, we first define an array of $(n-1)$ real variables denoted by DT in chronological order. By adding precedence constraints $\forall j \in \{0, 1, .., (n-2)\}$, $dt_j \leq dt_{j+1}$, we divide the time line into $(n-1)$ stages denoted by T. Every state variable is projected on stages as a sequence of variables $\{V_{p0}, V_{p1}, \ldots, V_{p(n-2)}\}$. To indicate the existence of each episode in each stage, we define an $m \times (n-1)$ Boolean variable matrix A, and add the constraints $(a_{ij}) \leftrightarrow (e_i^{\vdash} \leq t_j) \wedge (t_{j+1} \leq e_i^{\dashv})$. Then, we add the following constraints $\wedge_{i=0}^{m-1}(a_{ij} \rightarrow C_i)$, where C_i is the constraint in $i\text{-}th$

episode. If an episode is constant, we add equality constraints over its scoped state variables. To force each episode to happen in at least one stage, we add the constraints $\sum_{k=0}^{n-2} a_{ik} \geq 1$.

While the problems expressible as Time-insensitive ECSPs could also be written in more general CP formulations, we have shown that the encoding is nonintuitive when the time line is continuous and C_i can be any predicate in [1]. In the next section, we also show that the explicit declaration of temporal constraints allows precedence relations to be leveraged for more efficient solution algorithms.

3 Time-insensitive ECSP-Solver

In this section, we introduce an algorithm to solve Time-insensitive ECSPs. We walk through the solution process for the example problem in Figure 1 as follows.

The key to solving the example problem is to process the first task after the machine is ready. To find this solution, our approach generates a total order of the events, solves a CSP for every set of adjacent time windows with overlapping constant episodes, and continues generating total orders until one is found where a solution exists to every CSP. In order to efficiently generate the total orders, our algorithm exploits precedence relations, checks temporal consistency before solving the CSPs, and learns conflicts from both the temporal and state consistency checks.

The precedence relations from the example problem are that events $Start$, $Ready$, $Start$-2, End-2, and End occur in that order, and $Start$, $Start$-1, End-1, and $Start$-2 occur in that order. Already, this knowledge has reduced the number of feasible total orders to three — whether the first task happens before, across, or after $Ready$. Assume the first total order tested is that the first task happens before $Ready$. On checking the state consistency of this with a CSP solver, we immediately note that the state constraints for $Rest$ and $Task$-1 are incompatible. From this, we learn the conflict that the episodes $Task$-1 and $Rest$ cannot overlap. This allows us to skip the "across" total order and immediately generate the total order where the first task occurs after $Ready$.

The ECSP-Solver shown in Algorithm 1 consists of a decomposition module followed by three interacting layers: the ordering layer, the temporal layer, and the state constraint layer. The decomposition module exploits the temporal structure of the input ECSP and outputs a set of decomposed ECSPs (Line 1). The ordering layer is the top layer that takes as input a set of precedence relations (i.e., partial orders), and outputs a total order of all the events incrementally (Line 8). Considering both this total order and the existing temporal constraints (Line 9), the temporal layer checks the temporal consistency (Line 10), and returns an ordered instance that is also temporally consistent (Line 13). The state constraint layer decomposes and translates this ordered instance into several CSPs to be checked for consistency (Line 11). The ECSP-Solver incrementally generates total orders and checks both the temporal consistency and the state consistency under this order, until a consistent plan is found. In addition, inconsistent or consistent translated CSPs will be recorded and stored concisely to avoid unnecessary state consistency checks (Line 11).

Algorithm 1: $ECSP\text{-}Solver$

```
   Input  : I // ecsp instance
   Output: solution
1  S = {I_1, I_2...I_n} ← DecomposeECSP(I)
2  orders ← ∅
3  for I_i ∈ S do
4      Initialize ITO, STN
5      ← ⟨ ⟩
6      CC ← ⟨∅, ∅⟩ // CC = ⟨conflicts, cubes⟩
7      repeat
8          ⟨O, Δ_O, Q_conflict⟩ ← NextJump(ITO, Q_conflict)
9          STN.update(Δ_O)
10         if T-Consistent?(STN) then
11             ⟨cft, CC⟩ ← S-Consistent?(O, I_i, CC)
12             if cft = ∅ then
13                 orders.append(O) and break
14             else
15                 Q_conflict.update(cft)
16         else
17             Q_conflict.update(∅)
18     until Δ_O = ∅ and then return Failure
19 I ← orderECSP(I, orders)
20 return solution ← CSP-Solver(I)
```

3.1 Decomposition

Rather than ordering all the events, solving every part of the problem respectively and unifying the solutions are more effective. $DecomposeECSP$ shown in Algorithm 2 provides such a method that exploits the temporal structure and outputs decomposed problems. The first step is to project the problem into a STN (Simple Temporal Network) by removing all the state constraints (Line 1). We translate this STN into a distance graph in All-Pairs Shortest Path (APSP) form, which exposes implicit temporal constraints between event pairs (Line 2) [4]. We can construct a DAG (Directed Acyclic Graph) as a precedence graph, where precedence relations are determined by negative edges (Line 3).

‘‘

Fig. 2: Decompose at the event $Start$-2

.

Based on this precedence graph, an event of which any other event is either a predecessor or successor will be found as the candidate break point for decomposition, which is the necessary condition of decomposition (Line 5). We keep searching candidates until all candidates are exhausted or a break point is found whose predecessors

and successors have no connection (e.g., event $Start$-2 in Figure 2) (Line 8). Specifically, no connection equals the conjunction of two conditions: first, no constant episode is across these two event sets; second, in the MDN (Minimal Dispatchable Network) form of the STN, no temporal constraint is across these two event sets. We use the filter algorithm introduced in [11] to compute MDNs (Line 4). Finally, the STN is divided into two STNs with respect to the found break point, and the episodes in the original ECSP are projected back to each STN (Lines 9-10). As constant episodes are guaranteed to not cross two STNs, we project an across uniform episode by dividing it into two episodes connected to the break point. The decomposition is recursive and ends until no break point node is found.

Algorithm 2: $DecomposeECSP$

Input : I // ecsp instance
Output: S // an array of decomposed ECSPs

1 $STN \leftarrow GetSTN(I)$
2 $N_{apsp} \leftarrow GetAPSP(STN)$
3 $R_{precedence} \leftarrow GetPrecedenceRelations(N_{apsp})$
// Minimal Dispachable Network
4 $N_{mdn} \leftarrow GetMDN(N_{apsp})$
// $event_D$ succeeds E_L and precedates E_R
5 **if** $event_D \leftarrow FindDEvent(R_{Precedence})$ **then**
6 $E_L, E_R \leftarrow DivideEvents(STN, event_D)$
7 // No $Edge$ in N_{mdn} or $Constant\ Episode$ in I connecting E_L and E_R
8 **if** $NotConnect(E_L, E_R)$ **then**
9 **for** $I_i \in ProjectECSP(E_L, E_R, event_D, I)$ **do**
10 $S.push(DecomposeECSP(I_i))$
11 **return** S
12 **return** I

3.2 Ordering Layer

The ordering layer is the top layer of the ECSP-Solver which reuses the precedence relations from the decomposition module, and outputs an ordered ECSP. Ordering can help determine which episodes impose together at certain stage that is the interval between adjacent events, and decompose the original ECSP into a STN and a set of CSPs. It is complicated to consider highly coupled state constraints and temporal constraints at the same time, especially when continuous time is involved. As checking state and temporal consistency is usually more expensive than checking ordering consistency, our strategy is to prune infeasible orders in the first place.

The core of the ordering layer is Incremental Total Order (ITO) [16], which incrementally generates all the total order variations of the events that satisfy all the precedence relations and resolve ordering conflicts. ITO builds upon the algorithm to

generate linear extensions [13], which takes as input a set of partial orders and return all possible total orders. Since the number of total orders grows exponentially with respect to the number of events, ITO only stores one total order at each time and obtains the next order by swapping two adjacent events. If some partial order is violated, ITO backtracks, which prunes a portion of inconsistent total orders. The ITO search tree of $\{A, B, C, D\}$ with the partial orders $\langle A, C\rangle$ and $\langle B, D\rangle$ is shown in Figure 3.

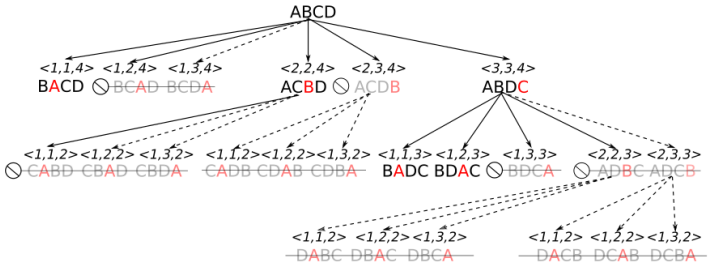


Fig. 3: ITO search tree of $\{A, B, C, D\}$ with the partial orders $\langle A, C\rangle$ and $\langle B, D\rangle$: ITO populates the search queue with labels $\langle i, j, level\rangle$. i is the previous index (i.e., the event index in the parent layer) of the event that is to be swapped, and j is the current index of the event that is to be swapped. $level$ is the index of the event that was swapped in the parent layer, and constraints the maximum search depth from the current node.

In Figure 4, we consider the ITO search tree of the example in Figure 1. As shown in the second line of Figure 4, ITO attempts to swap $Start$-1 and End-1, but it violates the partial order $\langle Start\text{-}1, End\text{-}1\rangle$. Then, ITO backtracks and stops trying to swap these two events during the next several iterations. As a result, ITO generates all three consistent orders from all the possible $5!$ variations within five swaps.

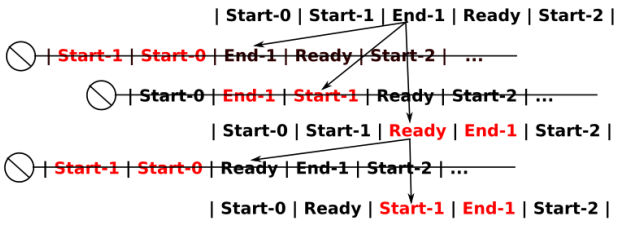


Fig. 4: ITO search tree of the scheduling example

As shown in Algorithm 3, the ordering layer is also conflict-aware. Given the conflict returned by the state constraint layer, our algorithms are guided to fast move within the ITO tree, until a total order that resolves this conflict is found. For example, in Figure 4, our algorithms can realize the inconsistency is because of scheduling the first task

before the machine is ready, and will directly jump to the consistent order that schedules the first task after the event $Ready$. We will explain how to achieve this feature when introducing the state constraint layer.

Algorithm 3: $NextJump$

Input : $ITO, Q_{conflict}$ // $Q_{conflict}$ is the corresponding conflict queue
Output: O, Δ_O,$\mathrm{Q}_{conflict}$//$eventorder, orderchangeandconflictque$

```
while ⟨order, swap⟩ ← ITO.NextOrder() do
    Δ_O ← UpdateOrderChange(Δ_O, swap)
    Q_conflict ← UpdateConflictQueue(Q_conflict, swap)
    if ResolveConflict?(swap, Q_conflict) then
        return ⟨O, Δ_O, Q_conflict⟩
return ⟨∅, ∅, Q_conflict⟩
```

3.3 Temporal Layer

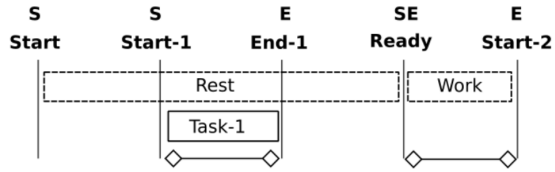
Ordering all the events imply adding a set of temporal constraints with the interval $\langle \epsilon, \infty \rangle$ on pairs of events, where ϵ is a small quantity. As a result, the modified STN is squeezed and the temporal consistency may not be maintained, so we have to check it when the order is changed. In our practical tests, temporal inconsistency happens infrequently, but checking temporal consistency for every total order is necessary for the soundness of our solver.

Since we obtain a sequence of total orders by changing the orders of a small portion of the events, most of the temporal constraints of the STN remain. Thus, it would be wise to reuse the search dependency and conflicts from checking the previous STNs. Our algorithm uses Incremental Temporal Consistency (ITC) [14], which is capable of updating the previous STN with incremental or decremental changes, and continuing the consistency checking.

3.4 State Constraint Layer

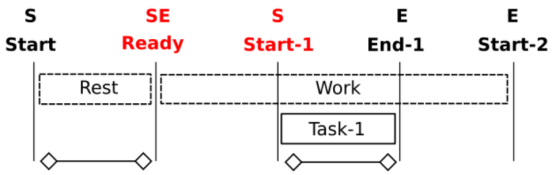
If the STN of an ordered ECSP is consistent, the final step is to check its state consistency by using Algorithm 4. As mentioned in the last section, we already decompose the temporal aspect and the state aspect of the ECSP by ordering the events. When it comes to checking the state consistency, the approach of considering the ordered ECSP as a single CSP is complete but not efficient. Because of the presence of the constant episodes, which force their scoped state variables to keep constant during covered stages, and results in cross-stage interaction, the approach to independently check the state constraints in each stage is unsound.

Instead of considering the ordered ECSP as a single CSP, we decompose it into several CSPs. In contrast to the constant episodes, the uniform episodes constrain the scoped state variables at every single time point, but the assigned values at different



(a) ECSP under $\langle Start, Start\text{-}1, End\text{-}1, Ready, Start\text{-}2 \rangle$.

(b) Swap $End\text{-}1$ and $Ready$.



(c) Swap $Start\text{-}1$ and $Ready$.

Fig. 5: Ordered ECSPs

time points do not affect each other. Thus, when two stages are not constrained by the same constant episodes, we check their consistency independently. For example, in Figure 5a and Figure 5c, each stage can be considered independently. However, in Figure 5b, the stage between $Start\text{-}1$ and $End\text{-}1$ need to be checked together, because they are covered by the same constant episode $Task\text{-}1$. To efficiently check state consistency, we prune redundant CSPs while maintaining the state consistency. Note that state constraints in one stage can be a superset of state constraints in other stages. In this case, the consistency of the superset stage imply the consistency of its subset stages, a problem is compacted by only including superset stages but still sound. For example, in Figure 5a, the stage between $Start\text{-}1$ and $End\text{-}1$ contains all the state constraints of its neighbors, thus its two neighbors are pruned and only the marked stages are considered. We call a set of connected superset stages as a Main Component (MC).

To introduce how to extract MCs, we first label the events according to whether they start or end episodes. There are three labels: S means the event starts some episode (e.g., $Start\text{-}1$); E means the event ends some episode (e.g., $End\text{-}1$); A SE event start some episode and end other episodes (e.g., $Ready$)[2]. The algorithm to extract MCs (Line 1) is linear, starting from the earliest event to the latest event. When we find an event that

[2] We do not consider events neither starting or ending any event, since they impose on effect on state consistency.

ends some episode (i.e., E or SE events), and some episode is started between this found event and the last S or SE event, the current stage ended by this found event is pushed as an element to a set. Intuitively, we record every local maximum of the episodes along the ordered events. When a stage is pushed and does not connect the next MC through constant episodes, the current set is recorded as a MC and another empty set is initialized for recording the next MC. For example, in Figure 5b, the stages between $Start$-1 and End-1 are recorded as a MC.

MC Relaxed	$S \leftrightarrow E$	$S \leftrightarrow SE$	$SE \leftrightarrow E$
MC Tightened	$E \leftrightarrow S$	$SE \leftrightarrow S$	$E \leftrightarrow SE$
No Change	$S \leftrightarrow S$	$E \leftrightarrow E$	
MC Changed	$SE \leftrightarrow SE$		

Table 1: Order change effects to Main Components (MCs)

We determine the state consistency of the ECSP by translating each MC to a CSP, and checking each translated CSP independently. To prune inconsistent CSPs, we leverage two ideas from prior work: ordering conflicts and ordering cubes. While the original *conflicts* denote inconsistent conjunctions of clauses [17], *cubes* were first introduced in [18] to represent consistent conjunctions of clauses. An inconsistent MC will be recorded as an ordering conflict, and a consistent MC will be recorded as an ordering cube, given as a set of end events of the stages in the corresponding MC. We have two mechanisms to leverage conflicts and cubes.

Algorithm 4: $S\text{-}Consistent?$

Input : O, I, CC // $CC = \langle conflicts, cubes \rangle$
Output: $conflict, CC$

$MCs \leftarrow decomposeOrderedECSP(I, O)$
for $MC \leftarrow MCs$ **do**
  if *Any of* $CC.cubes$ *subsumes* MC **then**
    $CC.cubes.update(MC)$
  else if MC *subsumes any of* $CC.conflicts$ **then**
    $CC.conflicts.update(MC)$
    return $\langle MC, CC \rangle$
  else
    $csp \leftarrow MC2CSP(MC)$
    if csp *is consistent* **then**
      $CC.cubes.update(MC)$
    else
      $CC.conflicts.update(MC)$
      return $\langle MC, CC \rangle$
return $\langle \emptyset, CC \rangle$

The local conflict mechanism is described in Algorithm 4 (Lines 2-14). Before a new MC is translated to a CSP, the local mechanism compares this new component with the stored conflicts and cubes (Lines 3 & 5). If this MC is subsumed by a cube (i.e., subset of a cube), it is state consistent. If this MC subsumes a conflict (i.e., superset of a conflict), it is state inconsistent. When new cubes and conflicts are added, the pool is updated by filtering subsuming cubes and subsumed conflicts (Lines 4 & 7 & 11 & 14).

The other mechanism is called the across-layer conflict mechanism described in Algorithm 1, which interacts with the ordering layer to leverage ordering conflicts. We modify ITO to maintain a ITO search path that contains all the ancestor orders from the current order to the root order (Line 9). Meanwhile, the conflict queue $Q_{conflict}$ (Line 5) corresponds each order in the ITO search path with an ordering conflict, and generates an order resolving the first ordering conflict of $Q_{conflict}$ (Line 8). As shown in Table 1, swaps between different events can relax, tighten, change (i.e., the effects are unknown) or make no change to the state constraints of a MC. Based on these rules, $ResolveConflict?$ (Algorithm 3 Line 4) can filter the event swaps that tighten or make no change to the inconsistent MCs. Therefore, Algorithm 3 only returns swaps that change or relax the constraints of the inconsistent MC. There are three issues about maintaining $Q_{conflict}$. First, when a total order that satisfies the precedence relations does not resolve the ordering conflict of its parent order, it will not be returned from $NextJump$. Instead, this order is pushed as the head of search path, and the ordering conflict of its parent order is updated and pushed to $Q_{conflict}$ (Algorithm 3 Line 3). Second, when the generated total order leads to a temporally inconsistent instance, an empty set is pushed to $Q_{conflict}$ (Line 17). Third, when the ITO backtracks, $Q_{conflict}$ also pops elements correspondingly (Algorithm 3 Line 3), such that the head of the conflict queue represents the ordering conflict that the next swap is expected to resolve. For example, in Figure 6, the first order that schedules the first task before the machine is ready is output by the ordering layer, but leads to the inconsistency of the first MC in Figure 5a. Under the second order that schedules the first task across $Ready$, the swap of End-1 and $Ready$ satisfies the precedence relations, but tightens the MCs represented by $\{End\text{-}1\}$, by swapping an E event (i.e., End-1) and a SE event (i.e., $Ready$). Note that, the updated ordering conflict changes to $\{Ready, End\text{-}1\}$. This updated ordering conflict and its corresponding total order are also recorded. As the third order relaxes the inconsistent MC by swapping a S event (i.e., $Start$-1) and a SE event (i.e., $Ready$), the ordering layer outputs this order. Finally, we find the total order that completely moves the first task after the machine is ready.

4 Empirical Results

To test the performance of our solver, we randomly generated a series of Episodic CSPs to model the communication networks configuration with time-evolving requirements [3]. To configure a communication network, each network node needs to choose a successor node to bridge a link. All these links dynamically construct a loop, such that the coming network flows can route to the destination by simply iterating the successor of the current router. In this section, we demonstrate the efficiency of our algorithm, compared to the state-of-art CP solver, Gecode [5], and the state-of-art MILP solver,

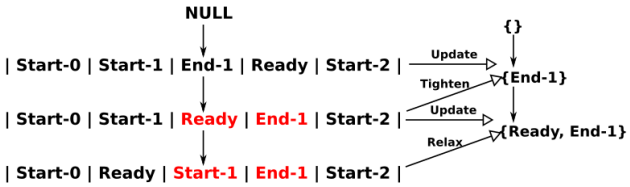


Fig. 6: Across-layer conflict mechanism

Gurobi [6], solving the equivalent formulations introduced in the section 3.3. As CP optimizer does not support continuous time line, we did not compare our solver to it in this section.

We simplified the requirements of loss, delay and bandwidth on links by requiring the absence or the presence of a certain link during an episode. This helped us to constrict the set of used constraint types, in order to evaluate solvers' performance when increasing the problem size. We modeled the successors of network nodes as state variables, the configuration requirements as episodes, and temporal relations between configuration requirements as temporal constraints. The requirement to construct the nodes as a loop was represented and simplified by a uniform episode with an all-different global constraint, scoping on all the nodes and the entire time horizon.

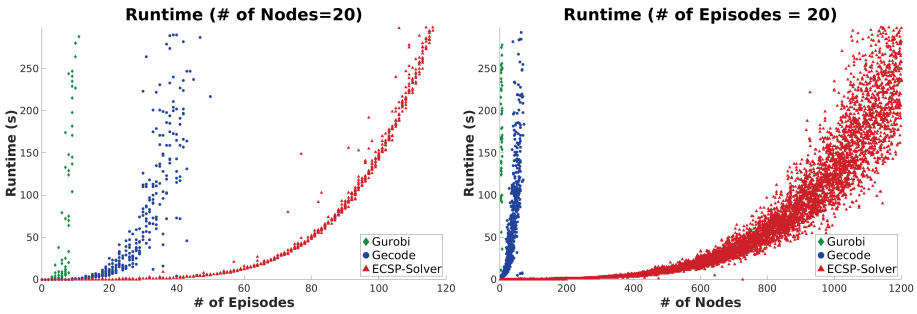


Fig. 7: Experiment results. Left shows when the number of nodes is 20. Right shows when the number of episodes is 20.

Using randomly generated network nodes, configuration requirements with time windows, and temporal constraints between configuration requirements, we evaluated the solvers' performance in two experiments. The experiments was to increase the number of episodes and nodes respectively with the step size 1 while fixing the other parameter. The timeout for each test was 300 seconds. In both cases, we generated 10 tests for each problem size and each solver, until the solver failed to solve the last 10 tests.

In Figure 8, the left figure shows the results of the first experiment. In all the tests, solvers configured 20 nodes under an increasing number of episodes (i.e., configura-

tion requirements with time windows). The ECSP-Solver could solve the problem with 120 episodes, while Gurobi and Gecode stopped at 10 episodes and 44 respectively. In the right figure of Figure 8, the second experiment increased the number of nodes while holding the number of configuration requirements at 20. As more nodes were added, the state constraint checking became more complex. While the solve time quickliy increase for Gurobi and Gecode, the ECSP-Solver could handle more complex state constraints with over 1000 nodes. This is because ECSP-Solver considers temporal relations before state constraints, avoiding unnecessary state constraint checking for inconsistent schedules. In addition, the state constraint layer can reason at the episode level to reduce MC consistency checking. With ordering conflicts, the ordering layer can also prune inconsistent total orders.

5 Conclusions

In many real world problems, a decision maker must decide on a course of actions, and the timing of actions, under constraints imposed by system dynamics, to meet a set of user-specified requirement specifications. Constraint programming has been a widely adopted paradigm for automated decision making in such problems. Recent work in CP has been extremely successful at solving systems with general classes of novel constraints, while the STN family of representations has been shown to be amenable to efficient specialized algorithms for handling temporal constraints. However, there has not being a formalism which combines explicit declaration of temporal constraints with novel constraints available to CP practitioners.

In this paper, we have motivated the introduction of problem representation which allows natural declaration of durative constraints over state trajectories. It is more natural to declare problems which prominently feature temporal constraints in such a formalism. Further, the explicit declaration of temporal constraints allows us to leverage results from the STN literature to decompose the problem into smaller independent problems for divide-and-conquer approaches.

We have thus introduced Episodic Constraint Satisfaction Problems, which extend the classical CSPs with the notion of time. ECSPs have been proven to be useful to represent scheduling problems with general state constraints and continuous time. Taking the Time-insensitive ECSPs as input, ECSP-Solver is able to schedule hundreds of tasks with thousands of state constraints, while maintaining relatively flexible schedules for successful execution.

However, the ECSP-Solver can be improved in two aspects. First, the ECSP-Solver cannot solve Time-sensitive ECSPs. Second, the conflicts at the constraint level and the the previous propagation dependencies are not used to accelerate solving another MC. In the future, we plan to explore a tractable set of Time-sensitive ECSPs (e.g., allocating a certain number of resources during a continuous time range), and use context-switching methods to incrementally check the MC consistency.

References

1. Becket, R.: Specification of flatzinc (2014)

2. Cesta, A., Oddi, A., Smith, S.F.: A constraint-based method for project scheduling with time windows. Journal of Heuristics 8(1), 109–136 (2002)
3. Chen, J., Fang, C., Muise, C., Shrobe, H., Williams, B.C., Yu, P.: Radmax: Risk and deadline aware planning for maximum utility. AAAI (2018)
4. Dechter, R., Meiri, I., Pearl, J.: Temporal constraint networks. Artificial intelligence 49(1-3), 61–95 (1991)
5. Gecode Team: Gecode: Generic constraint development environment (2006), available from `http://www.gecode.org`
6. Gurobi Optimization, I.: Gurobi optimizer reference manual (2016), http://www.gurobi.com
7. Hofmann, A.: Robust execution of bipedal walking tasks from biomechanical principles (2006)
8. Kim, P., Williams, B.C., Abramson, M.: Executing reactive, model-based programs through graph-based temporal planning. In: IJCAI. pp. 487–493 (2001)
9. Laborie, P.: Algorithms for propagating resource constraints in ai planning and scheduling: Existing approaches and new results. Artificial Intelligence 143(2), 151–188 (2003)
10. Levine, S.J., Williams, B.C.: Concurrent plan recognition and execution for human-robot teams. In: ICAPS (2014)
11. Muscettola, N., Morris, P., Tsamardinos, I.: Reformulating temporal plans for efficient execution. In: KR. pp. 444–452 (1998)
12. Nuijten, W., Le Pape, C.: Constraint-based job shop scheduling with iilog scheduler. Journal of Heuristics 3(4), 271–286 (1998)
13. Ono, A., Nakano, S.i.: Constant time generation of linear extensions. In: FCT. pp. 445–453. Springer (2005)
14. Shu, I.h., Effinger, R.T., Williams, B.C., et al.: Enabling fast flexible planning through incremental temporal reasoning with conflict extraction. In: ICAPS. pp. 252–261 (2005)
15. Timmons, E., Vaquero, T., Williams, B.C., Camilli, R.: Preliminary deployment of a risk-aware goal-directed executive on autonomous underwater glider. In: ICAPS. pp. 213–217 (2016)
16. Wang, D., Williams, B.: tburton: A divide and conquer temporal planner. In: Twenty-Ninth AAAI Conference on Artificial Intelligence (2015)
17. Williams, B.C., Ragno, R.J.: Conflict-directed a* and its role in model-based embedded systems. Discrete Applied Mathematics 155(12), 1562–1595 (2007)
18. Zhang, L., Malik, S.: Towards a symmetric treatment of satisfaction and conflicts in quantified boolean formula evaluation. In: International Conference on Principles and Practice of Constraint Programming. pp. 200–215. Springer (2002)