On the Efficient Synthesis of Deliberation and Reaction

Paper #28

Abstract

We introduce a formal framework for flexibly and efficiently synthesizing deliberation and reaction in a novel variation of the Sense-Plan-Act paradigm. The framework is based on partitioning the agent control structure into a collection of coordinated control loops and decoupling deliberation over goals from synchronization of agent state. We present algorithms for integrated agent control within this partitioned structure. Synchronization is a polynomial time multiple of the cost of primitive operations on a plan. We provide analytical and empirical results to support this complexity claim.

Introduction

Integrating deliberation and reaction has been a persistent challenge in the design of autonomous systems. The Sense-Plan-Act (SPA) paradigm for robot control embeds planning at the core of a control loop. Planning is typically the dominant cost and can limit the reactivity of an agent. Furthermore, the world can (and often does) change at a faster rate than the planner can plan. In this situation, the agent may thrash if the internal state of the plan gets out of synch with the actual state of the world. So, while SPA offers a general representational and computational framework for control, it is problematic for application in systems that require extensive deliberation and fast reaction.

The most common approach to resolve this dichotomy is the Three-layered architecture, e.g. The Remote Agent Experiment (**?**; **?**), ASE (**?**) and the LAAS architecture (**?**). These architectures employ heterogeneous representational and computational frameworks for deliberation and reaction. This approach is problematic in terms of integration and validation (**?**), motivating the introduction of unified frameworks like IDEA (**?**).

The contribution of this paper is to define a formal framework to partition the control responsibilities of an agent according to how far to look ahead (temporal scope), which state variables to consider (functional scope) and required reaction time (latency). Each partition is a control loop with its own internal SPA cycle, distinguished explicitly by these parameters with the agent as a whole employing a unified representational and computational model.

Although partitioning can be a powerful tool to reduce planning complexity it is unreasonable to require all planning to complete at the rate at which the environment is changing. Rather, planning within each control loop must be completed at the rate at which it must react to provide effective control. In order to allow a uniform quantization of time throughout the model, yet permit different rates of reaction for different control loops, it becomes necessary to allow sensing and planning to be interleaved. This allows multiple state updates to occur during deliberation, keeping the plan in synch with an evolving world state. To support this we decouple synchronization of the plan from deliberation over goals and allow them to be interleaved.

Within this framework, we present algorithms for integrating agent control and synchronizing agent state. Synchronization is a polynomial time multiple of the cost of primitive operations on a plan. We provide analytical and empirical results to support this complexity claim.

The paper is structured as follows. We begin with key definitions and concepts and then describe the partitioned agent structure and integrated control loop. We follow with a presentation of the synchronization problem in detail with algorithms for its solution. Results for experiments in simulation and in a real world environment are then described. A discussion and conclusions section terminates the paper.

Key Concepts & Definitions

The world, $\mathcal{W}$, contains a set of state variables, $\mathcal{S}_w$. An *agent*, situated in a world $\mathcal{W}$, is defined by:

- A set of **state variables**: $\mathcal{S} \subseteq \mathcal{S}_w = \{s_1, \ldots, s_n\}$.
- A set of **reactors**: $\mathcal{R} = \{r_1, \ldots, r_n\}$. Each *reactor* manages a component of the agent control problem to accomplish goals while observing the evolution of the world.
- A **lifetime**: $\mathcal{H} = [0, \Pi)$ defining the interval of time in which the agent will be active. $\mathcal{H} \subseteq \mathbb{N}$.
- The **execution frontier**: $\tau \in \mathcal{H}$ is the elapsed execution time as perceived by the agent. This value increases as time advances in $\mathcal{W}$. The unit of time is called a *tick*.

During its lifetime an agent observes the evolution of the world through $\mathcal{S}$ as the execution frontier advances. $\mathcal{R}$ provides the capabilities required to project the world evolution, and perform actions to alter world state.

Our *agent* design is based on timeline representations. Timelines are used to represent the evolution of the state variables from $\mathcal{S}$ and can be used as a representation for planning and execution control (**?**).

Definition 1 *A timeline, L, is defined by:*

- *a* **state variable***:* $s(L) \in \mathcal{S}$.
- *a set of* **tokens** *assigned to this timeline:* $\mathcal{T}(L)$. *Each token* $t \in \mathcal{T}(L)$ *expresses a constraint on the value of s over some temporal extent. A token is given by* $p(start, end, \overrightarrow{x})$ *with:*
  - *a* start *variable of domain* $\mathbb{N}$.
  - *an* end *variable of domain* $\mathbb{N}$. *The token holds for all ticks in [start, end).*
  - p *a predicate identifier*
  - $\overrightarrow{x}$ *a vector of the token parameters domains.*
- *an ordered set* $\mathcal{Q}(L) \subseteq \mathcal{T}(L)$ *of tokens. This set include tokens of* $\mathcal{T}(L)$ *that are temporally ordered.*

We use the notation, $L(t)$ to refer to the set of the tokens *ordered* in the timeline L that overlaps time t:

$$L(t) = \{a; a \in \mathcal{Q}(L) \land a.start \leq t < a.end\} \tag{1}$$

Elements of $L(t)$ are ordered to reflect their relative position in $\mathcal{Q}(L)$.

Partitioning

Partitioning exploits a divide-and-conquer principle by defining an agent as a composition of coordinated controllers. An agent is partitioned if it has more than one *reactor*. Reactors can deliberate and react at different rates, over different time horizons, and on different state variables. The agent coordinates the execution of reactors within a global control loop based on the SPA paradigm. The foundation for coordination is an information model that specifies a division of authority for determining the value for each state variable modeled in the agent over a designated temporal scope.

The Agent Information Model

Reactors may *own* or *use* one or more state variables. If a reactor *owns* a state variable it has the unique authority to determine the value for that state variable as execution proceeds. Every state variable has *exactly* one owner reactor:

$$\begin{aligned}&\forall s \in \mathcal{S}, \exists r_1 \in \mathcal{R}, \forall r_2 \in \mathcal{R} : \\ &\quad owns(r_1, s) \land owns(r_2, s) \Rightarrow r_2 = r_1\end{aligned} \tag{2}$$

A reactor *uses* a state variable *owned* by another if it requires notification of changes (called *observations*) as they happen or if it requests values for a state variable to take on in the future (called *goals*). Intuitively:

$$\begin{aligned}&\forall r_1, r_2 \in \mathcal{R}, \forall s \in \mathcal{S} : \\ &\quad uses(r_1, s) \land owns(r_2, s) \Rightarrow r_1 \neq r_2\end{aligned} \tag{3}$$

A state variable *owned* by a reactor may be functionally dependent on a state variable it *uses*. Such a dependency occurs, for example, where one state variable is an abstraction of another. For such cases, we define a *dependency* relation $\rhd$ as follows:

$$\forall r_1, r_2 \in \mathcal{R} :$$

$$\big(\exists s \in \mathcal{S}; uses(r_1, s) \land owns(r_2, s)\big) \Rightarrow r_1 \rhd r_2 \tag{4}$$

$$(\exists r_3 \in \mathcal{R}; r_1 \rhd r_3 \land r_3 \rhd r_2) \Rightarrow r_1 \rhd r_2 \tag{5}$$

Definition 2 *A reactor r is a controller defined by:*

- *a* **latency***:* λ_r *is the maximum amount of time the reactor r can use for deliberation.*
- *a* **look-ahead***:* $\pi_r \leq \Pi$ *defines the temporal duration over which reactor r deliberates. When deliberation starts for reactor r, its planning horizon is:*

  $$h_r = [\tau + \lambda_r, \tau + \lambda_r + \pi_r) \tag{6}$$

  with $\pi_r \geq \lambda_r$
- *a set of* **internal** *timelines:* $\mathcal{I}_r = \{I_1, \ldots, I_k\}$. *Timelines in* $\mathcal{I}_r$ *refer to state variables reactor r* owns*:*

  $$\forall I \in \mathcal{I}_r : owns(r, s(I)) \tag{7}$$
- *a set of* **external** *timelines:* $\mathcal{E}_r = \{E_1, \ldots, E_l\}$. *Timelines in* $\mathcal{E}_r$ *refer to state variables reactor r* uses*:*

  $$\forall E \in \mathcal{E}_r : uses(r, s(E)) \tag{8}$$
- *a set of* **goal tokens***:* $\mathcal{G}_r$. *Goal tokens express constraints on the future values of a state variable:*

  $$\forall g \in \mathcal{G}_r, \exists L \in \mathcal{I}_r \cup \mathcal{E}_r : g \in \mathcal{T}(L) \land g.\,start \geq \tau \tag{9}$$
- *a set of* **observation tokens***:* $\mathcal{O}_r$. *Observation tokens express actual values of a state variable:*

  $$\forall o \in \mathcal{O}_r, \exists E \in \mathcal{E}_r : o \in \mathcal{T}(E) \land o.\,start \leq \tau \tag{10}$$

  $$\forall o \in \mathcal{O}_r, \exists r_2 \in \mathcal{R}, \exists I \in \mathcal{I}_{r_2} : I(o.\,start) = o \tag{11}$$
- *a* **model***:* $\mathcal{M}_r$. *The model defines the rules governing the interactions between values on and across timelines.*

To know all the external timelines on an agent that are referring to a given state variable s, we define the operator $\varepsilon(s)$ such as :

$$\varepsilon(s) = \{E : E \in \cup_{r \in \mathcal{R}} \mathcal{E}_r, s(E) = s\} \tag{12}$$

We consider a special set of reactors, $\mathcal{R}_w$, which are primitive in that they have no dependencies:

$$\forall r \in \mathcal{R} : \mathcal{E}_r = \emptyset \Rightarrow r \in \mathcal{R}_w \tag{13}$$

In practice, such primitive reactors encapsulate the exogenous state variables of the agent. We further define the model for the agent, $\mathcal{M}_w$, as the union of all models of each individual reactor:

$$\mathcal{M}_w = \bigcup_{r \in \mathcal{R}} \mathcal{M}_r \tag{14}$$

The Agent Control Loop

The *uses* and *owns* relations provide the necessary details to direct the flow of information between reactors. Informally, observations flow from each reactor that *owns* a state variable to all reactors that *use* it. This occurs during a process called *synchronization*. Goals flow from reactors that *use* a state variable to the reactor that *owns* it. This occurs through a process called *dispatching*. *Deliberation* is the process of planning an evolution of state variables from current values to requested values. Synchronization, deliberation and dispatching are steps of the core agent control loop. The agent executes the set of reactors, R, concurrently. Execution of the loop occurs once per tick. Each step begins at the start of a tick by dispatching goals from users to owners. The agent then synchronizes state across all reactors in $\mathcal{R}$ at the execution frontier, τ. If there is deliberation to be done, the agent will do so in atomic steps until the clock transitions to the next tick, or deliberation completes. Thus, deliberation can be pre-empted by a clock transition. Algorithm 1 shows the top level agent control loop.

Algorithm 1 The agent control loop

```
RUN($\mathcal{R}, \tau, \Pi$)
       // If the mission is over, quit
 1     if $\tau \geq \Pi$ then return ;
       // Dispatch goals for all reactors for this tick
 2     DISPATCH($\mathcal{R}, \tau$);
       // Synchronize. If no reactor left afterwards, quit
 3     $\mathcal{R}' \leftarrow$ SYNCHRONIZEAGENT($\mathcal{R}, \tau$); // Alg. 2
 4     if $\mathcal{R}' = \emptyset$ then return ;
       // Deliberate in steps until done or the clock transition
 5     $\delta \leftarrow \tau + 1$;
 6     $done \leftarrow \perp$;
 7     while $\delta > \tau \wedge \neg done$
 8         do $done \leftarrow$ DELIBERATE($\mathcal{R}', \tau$);
       // Idle till the clock transitions
 9     while $\delta > \tau$ do SLEEP;
       // Tail recursive, with possibly reduced reactor set
10     RUN($\mathcal{R}', \tau, \Pi$);
```

Note that τ is incremented independantly of the algorithm.

Synchronization

Synchronization occurs at τ and ensures a *complete* and *consistent* view of agent state at the execution frontier. In a partitioned control structure, opportunities for inconsistency exist since each reactor has its own representation for the values of a state variable. While we allow divergent views of future values of a timeline to persist, we require that all timelines converge at the execution frontier. For an agent to be complete, agent state variables must have a valid value at τ. It is the responsibility of the owner reactor to determine this value during synchronization and the responsibility of users of that state variable to reconcile their internal state with this observation. This explicit ownership specification resolves conflicts.

Requirements

The concept of a flaw as described in (**?**) is central to the definition of synchronization. Fundamentally a flaw is a potential inconsistency that must be resolved. We are concerned with flaws that may render the state of the reactor inconsistent at the execution frontier. More specifically, we are concerned with any token, *t*, that *necessarily* impacts any timeline *L* of a reactor *r* at the execution frontier τ which has not been *inserted* in $\mathcal{Q}(L)$. Formally a flaw is a tuple $f = (t, L)$ that is resolved by *insertion* in $\mathcal{Q}(L)$.

Definition 3 *A reactor r is synchronized for τ, denoted $Ⓢ(r, \tau)$, when the following conditions are satisfied:*

- *All flaws are resolved at τ when*
$$\mathcal{F}_\tau(r) = \emptyset \tag{15}$$
where $\mathcal{F}_\tau(r)$ returns an arbitrarily ordered set of flaws of a reactor for synchronization at the execution frontier. A formal definition of $\mathcal{F}_\tau(r)$ follows in Equation (29).
- *All internal and external timelines have a unique valid value at τ, i.e there are no "holes" or conflicts in the timeline:*
$$\forall L \in (\mathcal{I}_r \cup \mathcal{E}_r), \exists o \in \mathcal{T}(L) : L(\tau) = \{o\} \tag{16}$$
- *All external timelines have the same value as the corresponding internal timeline of the owner reactor at τ:*
$$\forall L \in \mathcal{E}_r, \exists r_2 \in \mathcal{R}, \exists I \in \mathcal{I}_{r_2} :$$
$$s(I) = s(L) \wedge I(\tau) = L(\tau) \tag{17}$$

The synchronization of the agent is simply expressed as the synchronization of its reactors:
$$\forall r \in \mathcal{R} : Ⓢ(r, \tau) \tag{18}$$

Assumptions

While synchronization can be formulated and solved as a distributed planning problem over a narrow temporal scope, we use the following set of assumptions to reduce synchronization complexity to polynomial time. Our approach builds on the semantics of the partitioned structure to enable synchronization of the agent via incremental local synchronization of each reactor.

The Monotonicity Assumption (MA) From the perspective of a reactor, observations are taken as facts that are exogenous and monotonic: once an observation is published/received, it cannot be retracted. This implies that observations should not be published by a reactor until it has been synchronized:
$$\forall r \in \mathcal{R}, \forall o \in \mathcal{O}_r, o.start = \tau :$$
$$\exists r_1 \in \mathcal{R}, owns(r_1, s(o)) \wedge \tag{19}$$
$$Ⓢ(r_1, \tau) \tag{20}$$

The Inertial Value Assumption (IVA) The last observation made on an *external* timeline is valid until a new observation is received. The implication is that once a reactor has received all the observations for its external timelines at τ, then:
$$\forall r \in \mathcal{R}, \forall E \in \mathcal{E}_r :$$
$$\exists o \in \mathcal{O}_r : o.start = \tau \Rightarrow E(\tau) = o \tag{21}$$
$$\nexists o \in \mathcal{O}_r : o.start = \tau \Rightarrow E(\tau).end > \tau \tag{22}$$

Notice that the above equation ensures that all values of an external timeline contain an observation up till τ. Consequently we require an initial observation at τ = 0. In the case of internal timelines, the model $\mathcal{M}_w$ must specify the value to assign in all cases. A corollary of the MA is that the IVA should not be applied for a reactor r_1 until:
$$\forall r_2 \in \mathcal{R}, r_1 \rhd r_2 : Ⓢ(r_2, \tau) \tag{23}$$

The Acyclic Dependency Assumption (ADA) In principle, two reactors could be interdependent, where each uses and owns state variables of the other. Were this the case, synchronization could require iteration over the reactors until quiescence. To avoid this, we assume the reactor dependency graph is acyclic:

$$\forall r_1, r_2 \in \mathcal{R} : r_1 \rhd r_2 \Rightarrow \neg(r_2 \rhd r_1) \qquad (24)$$

Consequently, all reactors are distributed across a directed acyclic graph (DAG) where the root nodes are the reactors of $R_{\mathcal{W}}$. The imposition of a DAG allows us to assume the existence of an operator $R[i]$ with $i \in \mathbb{N}$ that accesses all elements in $\mathcal{R}$ such that:

$$\begin{aligned} &\forall R[i], R[j] \in \mathcal{R} : \\ &\quad R[i] \rhd R[j] \Rightarrow i > j \qquad (25) \\ &\quad R[i] = R[j] \Rightarrow i = j \qquad (26) \end{aligned}$$

Together, the MA, IVA and ADA break down the problem of synchronization of an entire agent to the incremental process of synchronizing each reactor.

Synchronizing the Agent

Algorithm 2 shows how the agent is synchronized and is accomplished by synchronizing each reactor in the order defined by $R[i]$. It is possible that a reactor may fail to synchronize. Such a failure implies that no consistent and complete assignment of values was possible for its timelines, and usually indicates an error in the domain model. Under these conditions, the agent must remove the reactor as well as all its dependents from the control structure in order to satisfy the requirements for consistent and complete state. This failure mode offers the potential for graceful degradation in agent performance with the possibility of continued operation of reactors implementing safety behaviors.

Algorithm 2 Global agent synchronization

```
SYNCHRONIZEAGENT($\mathcal{R}, \tau$)
1  $\mathcal{R}_{in} \leftarrow \emptyset$;
2  $\mathcal{R}_{out} \leftarrow \emptyset$;
3  for $i \leftarrow 1$ to SIZE($\mathcal{R}$)
     // Get next reactor on dependency list
4        do $r \leftarrow \mathcal{R}[i]$;
5           if ($\exists r_{out} \in \mathcal{R}_{out} : r \rhd r_{out}$)
               $\vee \neg$SYNCHRONIZE($r, \tau$) // Alg. 3
            // If r cannot be synchronized exclude it
6              then $\mathcal{R}_{out} \leftarrow \mathcal{R}_{out} + r$;
7              else $\mathcal{R}_{in} \leftarrow \mathcal{R}_{in} + r$;
   // Return the reactors that are still valid
8  return $\mathcal{R}_{in}$;
```

Synchronizing a Reactor

Algorithm 3 describes synchronization of a reactor based on a plan representation similar to (**?**). Synchronization begins by applying the inertial value assumption to extend the current values of external timelines with no new observations according to Equation 22. If the active planning horizon h_r (Equation 6) contains τ then deliberation has taken longer than λ_r and the reactor will be *relaxed*. Similarly, if there is no complete and consistent refinement of the set of flaws at the execution frontier (see Algorithm 5), the reactor will be relaxed. The relaxation procedure decouples restrictions imposed by planning from entailments of the model and execution state by deleting the plan but retaining observations and committed values. Goals will then have to be re-planned in a subsequent deliberation cycle. After relaxation, a second attempt is made to complete the execution frontier. If this fails, synchronization fails and the reactor will be taken off line by the agent. If this succeeds, the reactor will iterate over its internal timelines, publishing new values to users. Finally, at every step of synchronization, a garbage collection algorithm is executed cleaning out tokens in the past with no impact to the present or future. Details of garbage collection and plan relaxation are outside the scope of this paper.

Algorithm 3 Single reactor synchronization

```
SYNCHRONIZE($r, \tau$)
    // Apply IVA to extend current observations
1   for each E $\in \mathcal{E}_r$
2         do if ($\nexists o \in \mathcal{O}_r \cap \mathcal{T}(E); o.start = \tau$)
3               then $E(\tau).end \leftarrow E(\tau).end \cap [\tau + 1, \infty]$;
    // Complete execution frontier
4   if ($\tau \in h_r \vee \neg$COMPLETE($r, \tau$)) // Alg. 5
5      then if ($\neg$RELAX($r, \tau$) $\wedge \neg$COMPLETE($r, \tau$)) // Alg. 5
6               then return $\bot$
    // Publish new state values to users of internal timelines
7   for each I $\in \mathcal{I}_r$
8         do $State \leftarrow I(\tau)$;
9            if $State[0].start = \tau$
10              then for each E $\in \varepsilon(s(I))$
11                       do $\mathcal{T}(E) \leftarrow \mathcal{T}(E) \cup State$;
12                          $\mathcal{O}(E) \leftarrow \mathcal{O}(E) \cup State$;
    // Clean out tokens in the past that havve no impact
13  GARBAGECOLLECT($r, \tau$);
14  return $\top$;
```

Resolving Flaws at the Execution Frontier We will now formally define the operator $\mathcal{F}_\tau$ (used in Definition 3) and describe its application in Algorithm 4. The components of this definition are:

- the temporal scope of the execution frontier which we define to include the current state (i.e. tokens that contain τ) and the prior state (i.e. tokens that contain $\tau - 1$).
- an operator $\mathcal{F}_\pi$ which returns the set of flaws for deliberation. Flaws in $\mathcal{F}_\pi$ should not be in $\mathcal{F}_\tau$. The intuition is to check if a token is a goal, or if it there is a path from the token to a goal in the causal structure of the plan.
- a unit decision operator $\mathcal{U}$ for a flaw, *f*, that excludes flaws that can be placed at more than one location around τ in

$\mathcal{Q}(f.L)$:

$$\mathcal{U}(f) \Rightarrow \left(\begin{array}{c} \forall q_1, q_2 \in (f.L(\tau-1) \cup f.L(\tau)) : \\ (q_1 \otimes f.t) \wedge (q_2 \otimes f.t) \Rightarrow q_1 = q_2 \end{array} \right) \tag{27}$$

where $\otimes$ indicates that two tokens can be *merged*:

$$\begin{aligned} & p_1(s_1, e_1, \overrightarrow{x_1}) \otimes p_2(s_2, e_2, \overrightarrow{x_2}) \Rightarrow \\ & \quad (p_1 = p_2) \wedge (s_1 \cap s_2 \neq \emptyset) \wedge (e_1 \cap e_2 \neq \emptyset) \wedge \\ & \quad (\nexists x \in \overrightarrow{x_1} \cap \overrightarrow{x_2}; x = \emptyset) \end{aligned} \tag{28}$$

Definition 4 *We can now define the operator $\mathcal{F}_\tau(r)$, returning an arbitrarily ordered set of flaws, as:*

$$\mathcal{F}_\tau(r) = \bigcup_{L \in \mathcal{E}_r \cup \mathcal{I}_r} \left\{ \begin{array}{ll} f: & f.t \in \mathcal{T}(f.L), f.t \notin \mathcal{Q}(f.L), \\ & f.t.\,start \leq \tau \leq f.t.\,end, \\ & f \notin \mathcal{F}_\pi(r), \\ & \mathcal{U}(f) \end{array} \right\} \tag{29}$$

To *insert* a flaw, *f*, if there is a token, $q \in \mathcal{Q}(f.L)$ for which $q \otimes f.t$ then $f.t$ is merged with q. Otherwise it is assigned to a new position in $\mathcal{Q}(f.L)$. Either operation may be infeasible which will indicate that no complete and consistent refinement of the current execution frontier is possible. (**?**) describes token insertion in a partial plan. Algorithm 4 describes a procedure for flaw resolution, or failing in the event any one flaw cannot be resolved.

Algorithm 4 Resolve flaws at the execution frontier

```
RESOLVEFLAWS($r, \tau$)
1  while ($\mathcal{F}_\tau(r) \neq \emptyset$)
2      do $f \leftarrow \mathcal{F}_\tau(r)[0]$;
3          if $\neg insert(f.t, \mathcal{Q}(f.L))$
4              then return $\bot$
           // insert modified $\mathcal{F}_\tau$. See Eq. (29)
5  return $\top$;
```

Completion Algorithm 5 utilizes RESOLVEFLAWS to complete synchronization. It begins by resolving all the available flaws. Resolution of the set of flaws is a necessary condition for completeness. However, it is not a sufficient condition for two reasons. First, it is possible that holes may exist in the internal timelines (application of IVA ensures that all external timelines are complete) that must be filled. Second, it is possible that the end time for the current token in an internal timeline I, is an interval. This must be restricted so that $I(\tau)$ returns a singleton after synchronization; see Equation (16). To address this we define a policy to complete an internal timeline under these conditions. For the first case, MAKEDEFAULTVALUE will generate a default value according to the model which is inserted to fill the hole. In the second case, the end time of the current value will be extended. These modifications may generate more flaws in turn. For example, tokens previously excluded from synchronization by $\mathcal{U}$ may now become unit decisions. Furthermore, additional rules in the model may now apply. Consequently, we invoke RESOLVEFLAWS again.

Algorithm 5 Completion of the execution frontier

```
COMPLETE($r, \tau$)
1  if $\neg$RESOLVEFLAWS($r, \tau$) // Alg. 4
2      then return $\bot$
   // Complete Internal Timelines
3  for each I $\in \mathcal{I}_r$
4      do v $\leftarrow I(\tau)$;
5          if v = $\emptyset$
               // Timelines that have no value at $\tau$
               // get a predefined default value
6              then INSERT(I, MAKEDEFAULTVALUE(I, $\mu_r, \tau$));
7              else $v[0].end \leftarrow v[0].end \cap [\tau+1, \infty]$;
8  return RESOLVEFLAWS($r, \tau$);
```

Complexity Analysis

We now consider the complexity of synchronizing an agent. The key result is that synchronization is a polynomial time multiple of the cost of primitive operations on a plan.

We assume the following operators are executed in amortized constant time:

- $\mathcal{F}_\tau$ The operator to obtain the sequence of flaws in the execution frontier.
- INSERT(L, t) The procedure to insert a token t in timeline L.
- MAKEDEFAULTVALUE(L, μ_r, τ) The procedure to generate a default token for timeline L.

We further assume:

- GARBAGECOLLECT is linear in the number of tokens in the past that have not yet been removed.
- RELAX is linear in the number of tokens in all timelines.
- Insertion of a token by merging with an existing token generates no new flaws.
- The costs of $\otimes, \cap, \cup$ as used in synchronization are bounded and negligible.

Consider the procedure RESOLVEFLAWS. This procedure is linear in the number of flaws, since for each flaw encountered, it is resolved by an insertion operation within amortized constant time with no backtracking. Assume a reactor *r* has $\mathcal{N}_r$ timelines. In the worst case, every timeline in a reactor will require a new value for the current and prior tick. Assume that in the worst case, every new value generates a flaw for every other possible position in all timelines in the execution frontier (i.e. $2\mathcal{N}_r$ -1 flaws per new value). This gives a maximum complexity for RESOLVEFLAWS of $2\mathcal{N}_r \times (2\mathcal{N}_r\text{-}1)$ or O($\mathcal{N}_r{}^2$). In the worst case, the procedure COMPLETE calls RESOLVEFLAWS twice. However, if simply refining the execution frontier, this does not change the cumulative number of flaws. Since iteration over the internal timelines is linear in a value $\leq \mathcal{N}_r$, we have a complexity of O($\mathcal{N}_r{}^2$) for Algorithm 5.

In the worst case, synchronization of a reactor (Algorithm 3) incurs the following costs:

- O($\mathcal{N}_r$) to complete external timelines

- O($\mathcal{N}_r{}^2$) to call COMPLETE the first time, which we assume will fail.
- O(P_r) to RELAX the plan where P is the number of tokens in the plan.
- O($\mathcal{N}_r{}^2$) to call COMPLETE the second time, which we assume will succeed.
- O($\mathcal{N}_r$) to publish observations
- O(H_r) to garbage collect where H is the number of tokens in the plan that have passed into history.

Since synchronization of the agent is accomplished by iteration over the set of reactors, without cycling, we can state the worst case time complexity for synchronization as:

$$O(\circledS) = O(\sum_{r \in \mathcal{R}} \mathcal{N}_r^2 + P_r + H_r) \quad (30)$$

Deliberation

Even if we do impose constraints on how deliberation is integrated into the control loop for each reactor, we do not define an algorithm for deliberation. The motives behind these are twofold. First, we want deliberation to be preemptable by synchronization. This is essential since *maintaining the integrity of agent state dominates deliberation* over goals. Moreover, we want to permit deliberation *cycles* that require more than one tick to complete (i.e. where $\lambda_r > 0$). To ensure deliberation can be pre-empted we require that the process be executable in incremental *steps*, where the longest duration of a single step is within the margin of error for the clock. This ensures a reactor does not miss a tick. Second, we want deliberation to be bounded. We bound deliberation with a maximum latency λ_r and a maximum lookahead π_r. Deliberation is either *active* or *inactive*. If deliberation is *inactive* at the beginning of the agent control loop, a new planning horizon, h_r is defined (Equation 6). The operator $\mathcal{F}_\pi(r)$ will utilize this horizon to return the set of flaws for deliberation. If $\mathcal{F}_\pi(r) \neq \emptyset$ then deliberation becomes active. If a reactor becomes *inactive* in the current tick, it will not be reconsidered for deliberation until the next tick. Deliberation may complete immediately, even if $\lambda_r > 0$. However, if it does not complete within λ_r *ticks*, it will be abandoned as described in Algorithm 3: line 4. Goals that are necessarily before h_r will be rejected.

Dispatching

The motivation for dispatching goals from one reactor to another is to enable efficient compositional control by sharing as much information as necessary but as little information as possible. Therefore, we exploit the latency and lookahead values of reactors to bound the amount of information to be sent from a *user* reactor to an *owner* reactor. If a token t is to be dispatched from a reactor r, the following conditions must hold:

- Deliberation is *inactive*. The justification for this requirement is that if planning is incomplete, then partial results should not be dispatched since they might not be safe or feasible.
- The token t is inserted on an external timeline of reactor r: $\exists E \in \mathcal{E}_r; t \in \mathcal{Q}(E)$
- The token t is in the dispatch window of the *owning* reactor for $s(E)$:

$$\exists q \in \mathcal{R}, owns(q, s(E)) :$$
$$t.\,start \cap [\tau + \lambda_q, \tau + \lambda_q + \pi_q] \neq \emptyset$$

Experimental Results

In this section we evaluate the performance of synchronization in partitioned and non-partitioned control structures. When plan operations are constant time, we show that synchronization is $O(N^2)$ in the worst case. Moreover, we demonstrate that actual costs of synchronization are accrued based on what changes at the execution frontier, making it efficient in practice. We further demonstrate that since operations on a plan are typically not constant time, but vary in diverse and implementation dependent ways according to plan size, partitioning control loops can substantially reduce the net cost of synchronization and deliberation. We also describe results from an implementation of the principles described in an embedded environment.

Lab Experiments

We implemented our framework using the EUROPA$_2$ (**?**) and (**?**) constraint-based temporal planning library. The foundation of our implementation is a Deliberative Reactor whose structure is shown in Figure 1. All timelines and tokens for a reactor are stored in the plan database. The planner uses chronological-backtracking refinement search to deliberate. The agent invokes each reactor as defined in the Algorithm 1. For our experiments, we define a single model shared by all reactors. It contains a timeline with a single token type given by the predicate $p(start, end)$. Each experiment is defined by the following problem parameters:

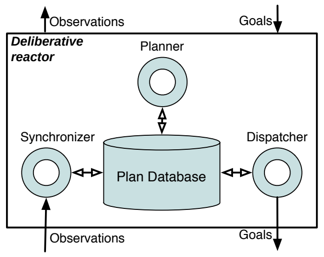


Figure 1: The Structure of a Deliberative Reactor

- I: The number of internal timelines per reactor.
- E: The number of external timelines per reactor. E is 0 for base reactors (i.e. $\mathcal{R}_w$).
- C: The number of relations to other timelines for any token. C is a measure of the connectedness between timelines and applies to internal timelines only.
- H: The agent horizon.
- D: The depth of the reactor control structure in the DAG hierarchy. A reactor with depth 0 is a leaf of the dependency graph. The duration of tokens on internal timelines of a reactor at depth d is given by 2^d.
- W: The width of the reactor control structure. There will be W reactors at the same depth in the dependency graph.
- π: The lookahead defined as a multiple of the durations of the tokens; the duration in turn were based on the depth of the reactors in the hierarchy.

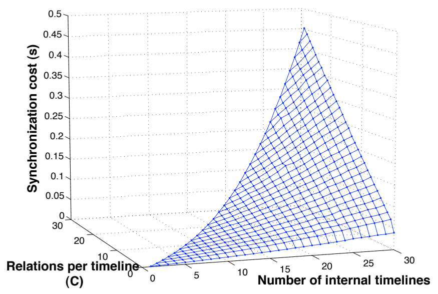


Figure 2: The relationship between synchronization time, number of constraints and number of internal timelines. W=1, D=0, E=0, H=50, $\pi = 0$, I $\in [1, 30]$ and C $\in [0, I-1]$

Our experiments were executed on a MacBook Pro running at 2 Ghz.

Figure 2 shows results of a run with a single reactor configuration where each problem instance runs for 50 ticks with W=1, D=0, E=0, H=50, $\pi = 0$. A set of problem instances are generated for combinations of I $\in [1, 30]$ and C $\in [0, I-1]$. There is no deliberation. CPU usage for synchronizing the agent is measured at every tick, and averaged over all ticks. The figure shows that average synchronization cost increases linearly in I and C and quadratically as the product of I and C.

An important impact of partitioning and information sharing is shown in Figure 3. We used a two reactor configuration where W=1, D=1, I=20, C=0, $\pi = 0$ and H=100 with E $\in$ [0 20]. Intuitively the cost of sharing information will increase with increasing number of external timelines. However, our results show that this cost increases *linearly* as a function of the size of the overlap for a constant C.

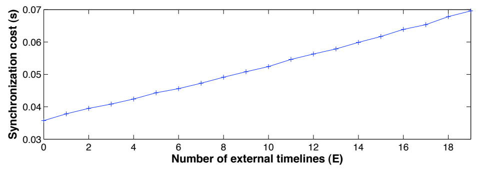


Figure 3: Information sharing via external timelines results in increased cost of synchronization linear in E. W=1, D=1, I=20, C=0, H=100,$\pi = 0$ and E $\in$ [0 20]

Figure 4 shows the impact of partitioning on problem solving, by varying the partitioned structure of an agent without changing the number of timelines being synchronized. We use 120 internal timelines spread evenly across all reactors. For each problem instance, C=0, E=0, H=10, D=0, $\pi = 10$ and W $\in$ [1 120] and I $\in$ [1 120] such that W $\times$ I = 120. When W=1, all timelines are in a single reactor and when W=120, there is only 1 timeline per reactor and the agent control structure is maximally partitioned. Deliberation fills out the timeline with 1 token per tick for π ticks with no search required. For each problem, the cumulative synchronization and deliberation CPU usage was measured. The number of state variables remains the same as timelines are apportioned between reactors showing the impact on problem solving by partitioning.

Figure 5(a) illustrates fluctuating synchronization costs with variation in the the rate of change of timelines in the agent. It also shows how partitioning impacts scalability.

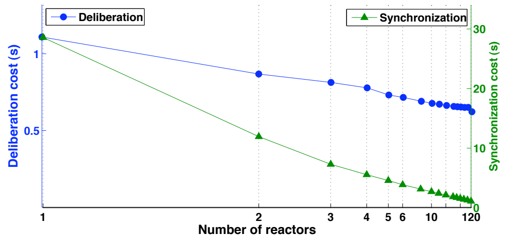
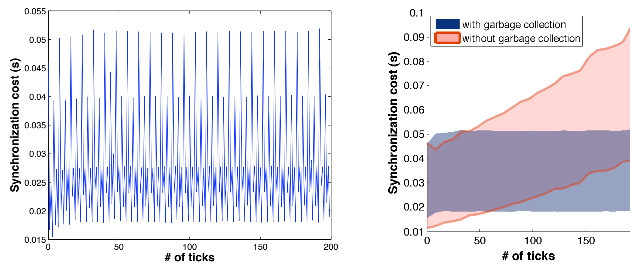


Figure 4: Synchronization costs associated with partitioning. C=0,E=0, H=10, D=0, $\pi = 10$ and W $\in$ [1 120] and I $\in$ [1 120] such that W $\times$ I = 120. All timelines are in a single reactor when W=1

Each reactor has ten internal timelines, and one external timeline with token duration of internal timelines set up to be 2^d where d is the depth of the reactor in its hierarchy. W=1, D=3, I=10, E=1, C=0, H=200 and $\pi = 0$. For each tick, the synchronization time is measured. Partitioning allows us to model the system so as to localize changes within a reactor to cap costs associated with synchronization. Further timelines at higher levels of abstraction necessarily change more slowly relative to timelines at lower levels of abstraction allowing partitioning to take effect. The figure also illustrates an important artifact in dealing with real-world system, namely the need to garbage collect the past. In the figure, one can see that the synchronization costs are trending upwards early on, and then leveling off. Garbage collection analyzes the executed plan and removes parts of the plan that have no impact on current and future states. As stated earlier, no details of garbage collection are presented in this paper. For purposes of illustration we show garbage collection turned off in Figure 5(b).



(a) Fluctuating synchronization costs with garbage collection

(b) Comparison with garbage collection off

Figure 5: Synchronization cost per tick

Sea Trials

Our framework has been applied onboard an Autonomous Underwater Vehicle (AUV) in coastal waters off Monterey Bay, California. The agent controlled the AUV for a number of missions, including those to detect, survey and sample dynamic ocean phenomenon over a specified area of more than 20 Sq. Km. The agent planned, dispatched and monitored all navigation and instrument control actions to accomplish a very high-level goal to conduct a volume survey. The deployed agent was a 3-reactor hierarchical configuration with 1 *tick* = 1 second. $\mathcal{R}[0]$ encapsulated the functional layer and owned 7 state variables ($\lambda_0 = 0, \pi_0 = 1$). $\mathcal{R}[1]$ handled navigation and instrument control, using all state variables of $\mathcal{R}[0]$ and owning 19 additional state variables ($\lambda_1 = 1, \pi_1 = 10$). Finally, $\mathcal{R}[2]$ handled high-level goal selection using 3 state variables of $\mathcal{R}[1]$ and owning 1 additional state variable ($\lambda_2 = 60, \pi_2 = 21600$). $\mathcal{R}[1]$ and $\mathcal{R}[2]$ were instances of a Deliberative Reactor sharing a single model and computational framework for all levels of abstraction and all time scales of deliberation and reaction. The agent ran on a 367 MHz EPX-GX500 AMD Geode stack using Red Hat Linux. The worst case cost for synchronization, involving a full relaxation of $\mathcal{R}[1]$, was 750 ms. The average case cost of synchronization was 120 ms. All deliberation timing limits were respected. Details are provided in (**?**), (**?**) and (**?**).

Discussion & Conclusion

The potential benefits of a unified framework that efficiently synthesizes deliberation and reaction within an SPA paradigm for robot control have been discussed extensively in (**?**; **?**; **?**). To realize these benefits we have focussed on improving the scalability of a unified approach by composition of the agent control structure through an explicit partitioning model. Within this framework, information flow and state synchronization is automatic. Furthermore, by introducing a clear distinction between synchronization of agent state and deliberation over goals, and permitting them to be interleaved, we allow a uniform representation in a single model, without requiring a uniform rate of reaction for control. This offers substantial flexibility in agent design.

We have implemented our framework and shown that synchronization, which is the core performance constraint of the architecture, is an O($\mathcal{N}^2$) multiple of the cost of primitive operations on a plan. We have demonstrated a partitioned agent control structure in real-world scenarios utilizing such a unified representational and computational. In conclusion, the following are worth noting:

- For weakly coupled reactors, partitioning can offer substantial performance improvements as we have shown. However this comes at the overhead of sharing information. Exactly where the cross over point occurs is a subject of further research. Techniques developed for partitioning CSPs such as (**?**) for example, may be relevant in this context.
- Plan failures can be localized. If a plan fails within a reactor, it is often possible to re-plan and recover without propagating the failure to observing reactor(s).
- Reactor failures can be localized. If a reactor fails to synchronize, or continually fails to plan, it will only impact its dependent reactors. This suggest a failure mode where system performance can degrade gracefully. The model can be structured to compel a reactor to take certain control actions under such circumstances.

Results for our real-world domain are very encouraging and we plan to extend the application of our framework to increasingly demanding control problems for ocean exploration.