A Process Algebra Genetic Algorithm

The MIT Faculty has made this article openly available. Please share how this access benefits you. Your story matters.


As Published: http://dx.doi.org/10.1109/tevc.2011.2160400

Publisher: Institute of Electrical and Electronics Engineers (IEEE)

Persistent URL: http://hdl.handle.net/1721.1/81446

Version: Author’s final manuscript: final author’s manuscript post peer review, without publisher’s formatting or copy editing

Terms of use: Creative Commons Attribution-Noncommercial-Share Alike 3.0
A Process Algebra Genetic Algorithm

Sertac Karaman Tal Shima Emilio Frazzoli

Abstract—A genetic algorithm that utilizes process algebra for coding of solution chromosomes and for defining evolutionary based operators is presented. The algorithm is applicable to mission planning and optimization problems. As an example the high level mission planning for a cooperative group of uninhabited aerial vehicles is investigated. The mission planning problem is cast as an assignment problem, and solutions to the assignment problem are given in the form of chromosomes that are manipulated by evolutionary operators. The evolutionary operators of crossover and mutation are formally defined using the process algebra methodology, along with specific algorithms needed for their execution. The viability of the approach is investigated using simulations and the effectiveness of the algorithm is shown in small, medium, and large scale problems.

Index Terms—Genetic algorithms, process algebra, UAV task assignment.

I. INTRODUCTION

Uninhabited aerial vehicles (UAVs) are becoming increasingly effective in performing missions that have previously been performed by manned airplanes. Their efficacy mainly stems from the lack of an on-board human operator. This enables development of systems with significant weight savings, lower costs, and allows performance of long endurance tasks. Currently, basic tasks of UAVs such as flying and trajectory planning from way-point to way-point can be automated. To enable the simultaneous cooperative operation of multiple such systems in complex missions, higher levels of autonomy are constantly sought. Within the last decade, cooperative control algorithms have been proposed to coordinate such multi agent systems in a preferably optimal way (see for example Refs. [1], [2], [3], [4], [5]).

Recently, cooperative control algorithms have been extended to handle more complex tasks and constraints, called the mission specifications, which are expressed using formal languages (see for example Refs. [6], [7]). These mission specifications include, but are not limited to, combinations of temporal ordering among tasks as well as conjunctive and disjunctive logical constraints. Specification languages with strict deadlines to specify and solve more complex UAV missions also have been considered (see, for example, Ref. [8]).

In Ref. [7], Process Algebra (PA) is used to specify a class of complex coupling constraints between tasks in UAV missions. This paper also adopts PA as the specification language for reasons to be outlined shortly. In computer science, process algebra is used for reasoning about the time behavior of software. For such reasoning, the software is assumed to be able to execute actions from a set \( A \) in some order. An action is a very general abstraction; it may refer to a blink of light, writing into a file, or moving a robotic arm. A behavior of the system is, then, a sequence of actions which are ordered with respect to the order that they were executed by the software. The sequence \( (a_1, a_2, \ldots, a_n) \) for which \( a_1, a_2, \ldots, a_n \in A \), for instance, would be a behavior of the system. Then, using the process algebra terms one can indeed specify the set of behaviors that a system can exhibit. This can be used as a design specification for automatic generation of software; or it can be used for checking whether a given software satisfies such a criterion.

In many military multiple-UAV missions, individual tasks like area search, classifying, or destroying a target, can be coupled with each other with temporal and logical constraints. Intuitively, these tasks, which we will refer to as atomic objectives, correspond to the actions, whereas the coupling constraints will be represented by the process algebra terms. High level tasks will be described using sets of atomic objectives, coupled through process algebra terms. We will denote such high level tasks as complex objectives. In the end, the entire UAV mission can be given as a single specification, i.e., a single complex objective.

The assignment of multiple cooperating UAVs to multiple tasks, such as collections of the atomic objectives mentioned above, requires the solution of a combinatorial optimization problem. The significant difficulty in solving many combinatorial optimization problems is that they are NP-hard and therefore cannot be solved in polynomial time by deterministic methods. So, due to the prohibitive computational complexity of the problem, the traditional deterministic search algorithms provide an optimal solution only for small-sized problems. For large sized problems they may provide a feasible solution in a given bounded run-time. Approximation algorithms can also be used for solving such problems. These algorithms give a solution of cost \( J \) to the problem with optimal cost \( J^* \) such that the ratio \( J/J^* \) is bounded by a known constant [9].

If optimality is not sought and the goal is to obtain a good feasible solution quickly, then stochastic search algorithms that employ a degree of randomness as part of their logic can be used. An algorithm of this type uses a random input to guide its behavior in the hope of achieving good performance in the “average case” and converge to a good solution in the expected runtime. Evolutionary algorithms (EA), which are inspired by the mutation selection process witnessed in nature, are common stochastic search methods used to solve many combinatorial optimization problems. These methods involve iteratively manipulating populations of solutions, termed chromosomes, that encode candidate good
solutions. The generational process is performed by applying evolutionary operators like selection, crossover, and mutation. Candidate solution selection is performed by evaluating the fitness (commonly chosen as inversely proportional to the cost) of each chromosome in the population. Historically, there were three main types of EAs: genetic algorithms (GAs), evolutionary strategies, and evolutionary programming, with GA being the most popular one [10].

Much work in applying GAs to combinatorial optimization problems is concerned with the encoding of the chromosomes and the use of special crossover operators that preserve the validity of the solution defined by the chromosome. The encoding and definitions of the evolutionary operators are problem specific. Recently, GAs have been proposed for solving UAV cooperative assignment problems [11], [12]. In Ref. [11], a GA was proposed for a scenario where a homogeneous set of multiple UAVs cooperate in performing multiple tasks (such as classify, attack, and verify) on multiple stationary ground targets. Solving such problems required assigning different tasks to different vehicles and consequently assigning each vehicle with a flyable path that it must follow. In Ref. [12], a GA was used to solve a cooperative UAV assignment problem where targets required simultaneous actions from several UAVs. In both of these studies simulation results showed the effectiveness of GAs in providing in real-time good quality suboptimal feasible solutions. Evolutionary algorithms have also been applied to related problems, such as the vehicle routing problem [13], [14].

Our work in this paper is mostly related to our previous work in Refs. [3], [6], [7], [8], [15], [11], [12], [16]. In Ref. [6], [8], [15], formal languages such as Linear Temporal Logic (LTL) and Metric Temporal Logic (MTL) were employed to describe complex tasks and constraints in military UAV operations. Although LTL and MTL are highly expressive specification languages, the algorithms presented in Ref. [6], [8] are limited to small problem sizes due to computational intractability of checking whether a specification given in LTL or MTL can be satisfied. To handle larger-scale problems more effectively, in Ref. [7], computationally more tractable process algebra specifications were incorporated into a tree search based task assignment algorithm (see Ref. [3] for the details of tree search). The computational efficiency of the algorithms tailored to handle process algebra specifications made their implementation on state-of-the-art UAV platforms possible. The algorithm proposed in [7] was recently demonstrated on a team of three UAVs in a joint U.S.-Australian military exercise in Australia [17]. Process algebra type specifications, if not as expressive as the temporal logics, were shown to describe a broad class of complex mission specifications of practical importance in Ref. [7]. Genetic algorithms, on the other hand, were used in Refs. [11], [12] to improve the computational effectiveness of the task assignment algorithms; however, their integration with complex mission specifications were never considered. This paper fills this gap by proposing a computationally effective algorithm, which can yet handle a broad class of complex tasks specified using process algebra.

The main contribution of this paper is a genetic algorithm solution to an assignment problem with high-level specifi-
right, most of the real-world systems execute a sequence of actions, which eventually lead to accomplishment of the high-level task and termination of the system. In the rest of the paper, we will assume that each high-level task, if it can be accomplished at all, can be accomplished by a terminating execution of the system, i.e., a finite sequence of actions. Such an execution is called a “behavior” of the system. It is important to note at this point that, most of the time, such a behavior that leads to successful fulfillment of the requirements is not unique, and it is crucial and challenging to naturally and formally “specify” the set of desired behavior of a system. Moreover, given the specification, designing algorithms that automatically enable the system to fulfill the specification is important and challenging in its own right.

Along with many other formalizations such as temporal logics [18], μ-calculus [19], or Petri nets [20], process algebra [21], [22], [23] is a methodology that can be used to specify the desired behavior of a system. Initiated and used in Computer Science to reason about computer software, process algebras found many applications in several diverse fields from web applications to hybrid systems [24]. Using process algebra for specification of UAV missions was first considered in Ref. [7]. This section presents an introduction to the process algebra based specification framework of Ref. [7].

An important notion in process algebra is the definition of the set of terms, which is formally given as follows:

**Definition III.1 (Terms of Process Algebra)** Given a finite set $A$ of actions, the set $\mathbb{T}$ of PA terms (defined on $A$) is defined inductively as follows:

- each action $a \in A$ is in $\mathbb{T}$;
- if $p, p' \in \mathbb{T}$ holds, then $p + p' \in \mathbb{T}$, $p \cdot p' \in \mathbb{T}$, and $p || p' \in \mathbb{T}$ also hold.

Each element of $\mathbb{T}$ is called a PA term, or a term for short.

Terms are related to each other as they can evolve from one to another. This evolution is made through a transition, denoted as $p \xrightarrow{a} p'$, where $p, p' \in \mathbb{T}$, and $a \in A$. This transition is read as “Process $p$ can evolve into process $p'$ by executing the action $a$”. There is also a special transition, denoted as $\sqrt{\cdot}$, which corresponds to the terminated process. By definition, the process $\sqrt{\cdot}$ has no actions to execute and cannot evolve into any other process.

Following the definition of the set of terms, each action can be a term such as $p = a$, where $a \in A$. Informally speaking, the system specified as $a$ has only one behavior: it can execute $a$ and then terminate, which is denoted as $a \xrightarrow{\sqrt{\cdot}} \sqrt{\cdot}$. To specify more complex systems, the PA specification framework offers the operators $(+)$, $(\cdot)$, and $(||)$, which are called the alternative, sequential, and parallel composition operators, respectively. Intuitively, a process that is specified with the term $p + p'$ behaves either like $p$ or $p'$, i.e., either executes a behavior of $p$ or executes one of $p'$ (but not both). The term $p \cdot p'$, on the other hand, first executes a behavior of the process $p$, and right after $p$ terminates, it executes a behavior of $p'$. The process $p \cdot p'$ is said to terminate when $p'$ terminates. The process $p || p'$ executes a behavior of each of $p$ and $p'$ concurrently. The process $p || p'$ is said to have terminated, when both $p$ and $p'$ terminate.

This informal presentation of the behavior of processes can be formalized by the operational semantics, which is defined as a set of transition system specifications (TSSs). A TSS is composed of a set $H$ of premises and a conclusion $\pi$, denoted as $\frac{\pi}{H}$, where $\pi$ is a transition and $H$ is a set of transitions. A TSS states that if the premises $H$ are possible transitions, then so is the transition $\pi$. The semantics (meaning) of each PA term is defined using the operational semantics of process algebra given as follows:

**Definition III.2 (Operational Semantics of PA)** The operational semantics of the process algebra is given by the following set of transition system specifications:

$$
\begin{align*}
\frac{a \xrightarrow{} \sqrt{\cdot}}{p} \\
p + p_1 \xrightarrow{} p + \sqrt{\cdot} &\quad p + p_2 \xrightarrow{} \sqrt{\cdot} \\
p_1 \xrightarrow{} p_1' &\quad p_2 \xrightarrow{} p_2' \\
\end{align*}
$$

where $a \in A$ and $p_1, p_1', p_2, p_2' \in \mathbb{T}$

Notice that the first TSS formally states that any action $a \in A$ can execute $a$ and then evolve to the terminated process, without requiring any other premises to hold. The next four TSSs provide the semantics of the alternative composition operator. Essentially, the second TSS states that if a process $p_1$ can execute an action $a$ and evolve to the terminated process, then so does the process $p_1 + p_2$ for any PA term $p_2$. The next three TSSs complement the semantics with other cases. The other TSSs in Definition III.2 provide the semantics of the sequential and parallel composition operators similarly.

With its recursive definition, the operational semantics associates each process with a set of traces that the process can execute. This set is merely the set of all behaviors of the system. More formally, any sequence $\sigma = (a_1, a_2, \ldots, a_k)$ of actions is called a trace of a term $p_0$ if and only if there exists a set of processes $p_1, p_2, \ldots, p_k$ such that $p_i \xrightarrow{a_{i-1}} p_i$ for all $i \in \{1, 2, \ldots, k\}$ and $p_k = \sqrt{\cdot}$. In other words, a trace of a process is a behavior of the process as it is a sequence of its available actions which lead to successful termination. Moreover, the set of all such traces of a process $p_0$, which will be denoted by $\Gamma_{p_0}$, formally defines the behavior of the process $p_0$.

**Example** Let $A = \{a_1, a_2, a_3, a_4, a_5\}$ be a set of actions. Consider the process $p_1 := a_1$ which can execute the action $a_1$ and terminate. Thus, $a_1 \xrightarrow{a_1} \sqrt{\cdot}$ is a legitimate transition, which yields the trace $\{a_1\}$. Notice that this is the only trace of $p_1$; hence, the behavior of $a_1$ is the set $\{(a_1)\}$, which includes its only trace.
Let us consider $p_2 := a_1 + a_2$. Notice that, for this process, $a_1 + a_2 \xrightarrow{a_1} \sqrt{}$ is a legitimate transition, since $a_1 \xrightarrow{a_1} \sqrt{}$ is also a legitimate transition (recall the second TSS in the operational semantics). Hence, $(a_1)$ is a trace of $p_2$. Furthermore, notice that $(a_2)$ is another trace. Thus, the behavior of $p_2$ is the set \[ \{(a_1), (a_2)\} \] of its traces.

Finally consider a larger example: $p_3 := (a_1 + a_2) \cdot (a_3 \parallel (a_4 \cdot a_5))$. Proceeding as above, the behavior of $p_3$ can be determined as
\[ \{(a_1, a_3, a_4, a_5), (a_1, a_4, a_3, a_5), (a_1, a_4, a_5, a_3), (a_2, a_3, a_4, a_5), (a_2, a_4, a_3, a_5), (a_2, a_4, a_5, a_3)\}. \]

Each process algebra term can be represented by a special data structure called a parse tree. The parse tree of a process algebra term is a binary tree composed of nodes, each of which encode either an operator or an action. The functions Operator and the action, respectively. While the function Operator is a single node which is labeled with an operator, every other node (this includes the root if the parse tree is not a single node) encodes an operator.

Given a term $p \in \mathcal{T}$ its parse tree is recursively defined as follows:

- If $p \in A$, i.e., $p$ is an action itself, then the parse tree of $p$ is a single node which is labeled with $p$.

- If $p = p_1 \triangleright p_2$, where $p_1, p_2 \in \mathcal{T}$ and has the parse tree of $p_1$ as its left and right children, respectively.

Example Consider the process $(a_1 + a_2) \cdot (a_3 \parallel (a_4 \cdot a_5))$. The parse tree of this process is presented in Figure 1. Notice that this parse tree is indeed a combination of the parse tree of the two processes $a_1 + a_2$ and $a_3 \parallel (a_4 \cdot a_5)$, bound with the sequential composition operator. Notice also that the former process has a parse tree formed by binding the parse trees of $a_1$ and $a_2$ with an alternative composition operator. The parse tree of $a_3 \parallel (a_4 \cdot a_5)$ can also be investigated with its subtrees, similarly.

As mentioned before, each node in the tree encodes either an operator or an action. The functions \( \text{Operator}_p(n) : \mathcal{N}_p \rightarrow \{\triangleleft, □\} \) and \( \text{Action}_p(n) : \mathcal{N}_p \rightarrow A \) return the encoded operator and the action, respectively. While the function \( \text{Action} \) is defined only for the leaf nodes, \( \text{Operator} \) is defined for all the other nodes in \( \mathcal{N}_p \). Finally, the function \( \text{Children}_p(n) \) maps each node to an ordered sequence of its children such that the left child is the first element of the sequence, whereas the right child is the last one.

IV. HIGH-LEVEL SPECIFICATION OF GENERIC OBJECTIVES

In this section, the process algebra framework is used to specify a class of vehicle routing problems. Although process algebra is not as expressive as other formalisms to express temporal logic constraints (e.g., Linear Temporal Logic), in this paper we choose process algebra for two reasons. First, process algebra offers computationally efficient algorithms, e.g., for checking whether a given string satisfies a given specification. This property allows us to design computationally efficient valid GA operators. Second, the hierarchical specification methodology of the process algebra allows building more complex specifications from simpler ones, which is illustrated with examples throughout this section.

Most of the material in this section is derived from that in [7], where the reader is referred to for a more thorough discussion in the context of UAV mission planning.

We consider a vehicle routing problem, in which a set \( \mathcal{O} \) of atomic objectives are assigned to a set \( \mathcal{V} \) of vehicles, so as to optimize a given cost function, while satisfying a specification given in the PA language. First, we define atomic objectives and employ process algebra to represent more generic objectives in terms of atomic ones. Then, we proceed with some preliminary definitions, followed by a formalization of the problem definition.

A. Objectives

Intuitively speaking, an atomic objective is a task that can not be represented by a combination of any others. In essence, atomic objectives are abstractions of individual tasks in a vehicle routing problem. In the context of, for instance, UAV mission planning, the first such abstractions were presented recently in Ref. [25] and further developed and employed in Ref. [7].

Definition IV.1 An atomic objective \( o \) is a tuple \( (x_0^o, x_0^e, T_0^e, v_o) \) where

- \( x_0^o \in \mathbb{R}^2 \) is the entry point
- \( x_0^e \in \mathbb{R}^2 \) is the exit point
- \( T_0^e \in \mathbb{R}_+ \) is the execution time
- \( v_o \in \mathcal{V} \) is a capable vehicle

Intuitively, only \( v_o \) can execute the atomic objective \( o \) in exactly \( T_0^e \) amount of time; moreover, \( v_o \) moves to the coordinate \( x_0^o \) to start executing \( o \) and ends up at coordinate \( x_0^e \) after the execution.

We will assume, without any loss of generality, that \( o \) can not be executed by any vehicle other than \( v_o \). Later in the
paper, we will show that the tasks that can be executed by one of several vehicles can be represented as a combination of different atomic objectives.

The definition of atomic objectives is quite general and can capture many different types of individual tasks for vehicle routing problems. In [25], Rasmussen and Kingston show that a similar abstraction can model, e.g., sector or area search, classification, attack, rescue, target tracking, reconnaissance, et cetera, in the UAV mission planning context.

Using the process algebra specification framework, more generic objectives, which impose temporal and logical constraints, can be composed from simpler ones. Given a set $O$ of atomic objectives, a (generic) objective is represented by a process algebra term $p$ defined on $O$ as the set of actions.

**Example** Let us present examples of atomic and generic objectives in the context of vehicle routing. Let us consider a scenario, in which a heterogenous set $\{v_1, v_2, v_3, v_4\}$ of four vehicles are bound to visit four cities, named $A$, $B$, $C$, and $D$, respecting the following constraints. The first and the second vehicles can only visit $A$, the third vehicle can visit only $B$, and the fourth vehicle can visit the remaining two cities, $C$, and $D$. The mission is to first visit $A$ by either $v_1$ or $v_2$, then obtaining a certain intelligence, e.g., surveillance data, from $A$, visit $B$, $C$, and $D$, in any order such that $C$ is visited before $D$ as some cargo has to be moved from $C$ to $D$ by the fourth vehicle.

This high-level specification of the vehicle-routing problem at hand can be described within the process algebra framework as follows. Let $O = \{o_1, o_2, o_3, o_4, o_5\}$ denote the set of atomic objectives. The atomic objectives $o_1$ and $o_2$ visit the city $A$, whereas the atomic objectives $o_3$, $o_4$, and $o_5$ visit the cities $B$, $C$, and $D$, respectively; the atomic objective $o_1$ can be executed by $v_1$, $o_2$ by $v_2$, $o_3$ by $v_3$, and $o_4$ and $o_5$ by $v_4$.

Notice that the constraint that the city $A$ must be visited either by $v_1$ or by $v_2$ can be represented by the process $o_1 \parallel o_2$. Similarly, the whole mission can be specified by the process algebra string $(o_1 + o_2) \cdot (o_3 \parallel (o_4 \cdot o_5))$.

For more examples of specifications of vehicle routing problems using the process algebra framework, we refer the reader to [7].

**B. Schedules, Observations, and Specifications**

A **single vehicle schedule** is a sequence of distinct pairs of atomic objectives and time instants. Intuitively speaking, a single vehicle schedule $\sigma_v$ for vehicle $v$ is a list of atomic objectives and their execution times to be executed by $v$. More precisely, if $(o, t) \in \sigma_v$ for some $o \in O$ and $t \in \mathbb{R}_+$, then the atomic objective $o$ is said to be scheduled to be executed at time $t$ by vehicle $v$. Note that the atomic objective and time pairs in a single vehicle schedule $\sigma_v$ are ordered according to their time component, i.e., $(o_i, t_i) <_{\sigma_v} (o_j, t_j)$ only if $t_i < t_j$ for all $(o_i, t_i), (o_j, t_j) \in \sigma_v$. A **complete schedule**, or schedule for short, is a set $S$ of single vehicle schedules that contains exactly one single vehicle schedule for each vehicle in $\mathcal{V}$.

Given two atomic objectives $o_i$ and $o_j$, let us denote the time it takes for vehicle $v$ to travel from the exit point of $o_i$ to the entry point of $o_j$ as $T^{o_i, o_j}_v$. Then, a complete schedule $\mathcal{P}$ is said to be **valid** if for each vehicle $v \in \mathcal{V}$ and for all pairs $(o_i, t_i) \in \sigma_v$ the atomic objective $o_1$ can indeed be executed at time $t_i$ by vehicle $v$. More precisely, for any $\sigma_v = \{(o_1, t_1), (o_2, t_2), \ldots, (o_k, t_k)\}$ in $\mathcal{P}$, the following holds: $t_{i-1} + T^{o_{i-1}, o_i}_v \leq t_i$ for all $i \in \{2, \ldots, k\}$.

A sequence $\pi = (o_1, o_2, \ldots, o_k)$ of atomic objectives is an **observation** of the schedule $S$ if the following holds: (i) any atomic objective $o$ that is scheduled in $S$ is an element of $\pi$, and (ii) for each atomic objective $o \in \pi$ there exists a time instance $t$ such that $t \leq t \leq T^o$, where $t$ is such that $(o, t) \in \sigma_v$ for some $v \in \mathcal{V}$, $T^o$ is the execution time of $o$, and (iii) for all we have $o_i, o_j \in \pi$, $o_i < \pi o_j$ if and only if $t_i \leq t_j$, where $t_i$ and $t_j$ are the time instances corresponding to $o_i$ and $o_j$, respectively. Intuitively, an observation is a sequence $\pi$ of atomic objectives such that corresponding to each $o_i$ that appear in $\pi$ one can find a time instance $t_i$ within the execution interval of the atomic objective $o_i$ so that the ordering of these time instances is the same as the ordering of their corresponding atomic objectives in $\pi$. From here on, we will denote the set of all observations of a valid complete schedule $S$ by $\Pi_S$.

Following the definition of observations, a specification and its satisfaction is formalized as follows.

**Definition IV.2 (Specification)** A **specification** is a process algebra term defined on the set $O$ of atomic objectives. A **valid schedule** $S$ is said to satisfy a specification $p$ if and only if any observation of $S$ is a trace of $p$, i.e., $\Pi_S \subseteq \Gamma_p$ holds.

**Example** Consider the scenario in the previous example. Recall that the specification was $p_{\text{spec}} = (o_1 + o_2) \cdot (o_3 \parallel (o_4 \cdot o_5))$. Consider the schedule that assigns $\sigma_1 = \{\sigma_{v_1}, \sigma_{v_2}, \sigma_{v_3}, \sigma_{v_4}\}$, where $\sigma_{v_1} = ((o_1, t_1), \sigma_{v_2} = \delta, \sigma_{v_3} = ((o_3, t_3), \sigma_{v_4} = ((o_4, t_4), (o_5, t_5))$. The time instances $t_i$ are depicted in Figure 2. Notice that this schedule has exactly three observations: $\pi_1 = (o_1, o_3, o_4, o_5), \pi_2 = (o_1, o_4, o_3, o_5)$, and $\pi_3 = (o_1, o_4, o_3, o_5)$ (see Figure 3 for depictions of the time instances $t_i$ that lead to these observations). Notice that all these observations are indeed traces of $p_{\text{spec}}$. Hence, $S$ satisfies $p_{\text{spec}}$.

![Fig. 2. A timeline of the example schedule.](image-url)
The completion time of the schedule \( S \) will be referred to as the completion time \( \tau \), and \( \tau_v = t' + T_{i'} \), where \( (o', t') = \sigma_v([\sigma o]) \). The real number \( \tau \) will be referred to as the completion time of \( \sigma_o \). Using the completion times \( \{\tau_v\}_{v \in V} \), it is possible to define the cost of the schedule \( S \) in many ways. Two of the common cost functions include the total completion time \( J_1 \) and maximum completion time \( J_2 \), which are defined, respectively, as

\[
J_1(S) = \sum_{v \in V} \tau_v, \quad J_2(S) = \max_{v \in V} \tau_v. \tag{1}
\]

The problem definition is given as follows.

**Problem IV.3**

Given a set \( V \) of vehicles, a set \( O \) of atomic objectives, traveling times \( T_{i,v_o,a_j} \) for all \( v \in V \) and all \( o_i, o_j \in O \), and a process algebra specification \( p_{spec} \) defined on \( O \), the optimal planning problem with PA specifications is to find a valid schedule \( S \) such that (i) \( S \) satisfies the specification \( p_{spec} \), and (ii) the cost function \( J_1(S) \) (or \( J_2(S) \)) is minimized.

Recently, a tree search based solution to a similar problem was given in [7], which extends the algorithm in [3] to handle process algebra specifications. The tree search algorithm presented in these references effectively searches the state space of all solutions and returns a feasible solution to the problem in polynomial time with respect to the size of the specification \( p_{spec} \) as well as the number of vehicles. Moreover, given extra time, the algorithm improves the existing solution with the guarantee of termination with an optimum solution in finite time. In the next section, we provide a genetic algorithm heuristic solution to Problem IV.3.

V. GENETIC ALGORITHM

Given a specification \( p_{spec} \), any trace of \( p_{spec} \) is a chromosome, usually denoted by \( X, X_1, X_2 \) et cetera. The genetic algorithm (GA) maintains a set \( X \) of chromosomes called the generation. The GA is initialized with a randomly created generation of chromosomes. At each iteration, (i) parent chromosomes are selected stochastically from \( X \) according to their fitness, (ii) new chromosomes are generated from their parents using the crossover operation, (iii) some of the chromosomes are generated by mutating the existing ones randomly, and (iv) the chromosomes that are more fit than others are carried to the next generation. In this section, we first present the details of these four evolutionary operators, after discussing the relationship between the schedules and the chromosomes. We also present the genetic algorithm solution as a whole and discuss its correctness.

A. The Relationship Between Chromosomes and Schedules

Each chromosome \( X_i \) corresponds naturally to a valid complete schedule denoted as \( S(X_i) \). Before formalizing the construction of \( S(X_i) \), let us introduce the following definition. An atomic objective \( o \) is said to be a predecessor of another atomic objective \( o \) in a specification \( p \) if the following two conditions hold: (i) there exists a trace \( \gamma \) of \( p \), in which \( o \) appears before \( o \) in \( \gamma \), i.e., \( o <_\gamma o \), (ii) there is no trace of \( p \), in which \( o \) appears before \( o \). Equivalently, it can be shown that \( o \) is a predecessor of \( o \) in \( p \) if and only if the parse tree of \( p \) includes a node that binds two terms \( p_1 \) and \( p_2 \) with a sequential composition operator as in \( p_1 \cdot p_2 \), where \( p_1 \) includes \( o \) and \( p_2 \) includes \( o \) as an atomic objective (see Ref. [7]). We will denote the set of all predecessors of a given atomic objective \( o \) in a specification \( p \) by \( \text{Pred}_p(o) \).

Notice that the sets \( \text{Pred}_p(o) \) for all \( o \in O \) can be formed efficiently by observing the parse tree of \( p \). This process can be executed (only once before starting the algorithm) in time that is bounded by a polynomial in the size of \( p \).

Given a chromosome \( X \), the complete schedule \( S(X) \) is generated recursively as follows:

- For a chromosome of the form \( X = o \), where \( o = (x_1^o, x_2^o, T_0, v_o) \), we define \( S(X) \) to be a schedule that assigns the atomic objective \( o \) to vehicle \( v_o \) to be executed at time \( t \), where \( t \) is the time required for \( v_o \) to travel from its initial position to the entry point \( x^o_1 \) of the atomic objective \( o \). More precisely, \( S(X) := \{ o_1, o_2, \ldots, o_N \} \), where \( o_{v_o} = (o, t) \), with \( t \) being the time it takes \( v_o \) to travel from its initial position to \( x^o_1 \), and for all \( v_j \neq v_o \) and \( v_j \in V \), \( o_{v_j} \) is an empty sequence. Note that \( N \) represents the number of vehicles, i.e. \( N = |V| \).

- Given a chromosome \( X \) with \( |X| = K > 1 \), let \( o = (x_1^o, x_2^o, T_0, v_o) \) be the last atomic objective that appears in \( X \), i.e., \( o = X(|X|) \), and \( X' \) be the sequence that is the same as \( X \), except that it does not contain \( o \); more formally, \( X' \) is such that \( X = X'(|X|) \) holds. Let us denote \( S(X') = \{ o_1, o_2, \ldots, o_K \} \). Then, the schedule \( S(X) \) is defined as \( \{ v_{o_1}, v_{o_2}, \ldots, v_{o_K} \} \), where \( v_{o_i} = o_{v_i} \) for all \( v_i \neq v_o \) and \( v_{o_i} \) is such that \( v_{o_i}(k) = o_{v_i}(k) \) for all \( k \in \{1, 2, \ldots, |o_{v_i}|\} \) and \( \sigma_{v_i}(o_{v_i}^i + 1) = (o, t) \). That is, \( S(X) \) is the same as \( S(X') \) except that, in addition, in \( S(X) \) atomic objective \( o \) is assigned to vehicle \( v_o \) to be executed at time \( t \). The execution time \( t \) is computed as follows. Recall that \( \tau_{o_{v_i}} \) denotes the completion time of the schedule \( o_{v_i} \). The execution time \( t \) is the smallest time that is greater than both \( \tau_{o_{v_i}} \) and the maximum

![Fig. 3](image-url)
execution time of any predecessor of \( o \) that is scheduled in \( S(X') \), i.e.,

\[
\tilde{t}_i = \max \{ \tau_{\sigma'_u}, \\
\max \{ t_0 | (\tilde{o}, \tilde{t}_0) \in \sigma'_u, \sigma_{u'} \in S(X'), \tilde{o} \in \text{Pred}_p(o) \} \}
\]

Given a chromosome \( X \) with \( |X| = K \), let \( X_1, X_2, \ldots, X_{K-1} \) be the sequences defined as follows: for all \( k \in \{1, 2, \ldots, K - 1\} \), we have \( |X_k| = k \) and \( X_k(i) = X(i) \) for all \( i \in \{1, 2, \ldots, k\} \). Notice that, algorithmically, \( S(X_1) \) can be computed easily, and \( S(X_k) \) can be computed using \( S(X_{k-1}) \). Hence, \( S(X) \) can be constructed recursively starting from \( S(X_1) \).

**Example** Consider the scenario in the previous example. Recall that the specification was \((o_1 + o_2) \cdot (o_3 \parallel (o_4 \cdot o_5)) \). It can be shown that \( o_1 \) and \( o_2 \) have no predecessors. However, \( o_3 \) and \( o_4 \) both have \( o_1 \) and \( o_2 \) as their predecessors. The atomic objective \( o_5 \), on the other hand, has \( o_1, o_2 \), and \( o_4 \) as its predecessor.

Traces of this process algebra term were presented in an earlier example. One of these traces was \((o_1, o_4, o_3, o_5) \). Algorithmically, the schedule corresponding to this chromosome is formed as follows. First, the atomic objective \( o_1 \) is handled: it is assigned to \( v_1 \), since \( v_1 \) is the capable vehicle for \( o_1 \); the time of execution \( t_1 \) is selected such that \( t_1 \) is the least time that, starting from its initial position, \( v_1 \) gets to \( A \). Next, \( o_4 \) is scheduled to be executed by its capable vehicle, \( v_4 \), at time \( t_4 \). Recall that \( o_1 \) is a predecessor of \( o_4 \). Hence, \( t_4 \) set after both the arrival time of \( v_4 \) to the city \( C \) as well as the completion time of \( o_1 \). Continuing this procedure similarly, \( o_3 \) is scheduled next to be executed by \( v_3 \) at time \( t_3 \), where \( t_3 \) is after both the arrival of \( v_3 \) to the city \( B \) and the completion time of \( o_1 \). Finally, \( o_5 \) is assigned to be executed at time \( t_5 \) such that \( t_5 \) is after the arrival of \( v_4 \) to \( D \) coming from \( C \) and after the end execution of all its predecessor atomic objectives, \( o_1, o_3, \) and \( o_4 \).

Hence, we have generated the complete schedule \( S = \{\sigma_{v_1}, \sigma_{v_2}, \sigma_{v_3}, \sigma_{v_4}\} \), where \( \sigma_{v_1} = ((o_1, t_1)) \), \( \sigma_{v_2} = \delta \), \( \sigma_{v_3} = ((o_3, t_3)) \), and \( \sigma_{v_4} = (o_4, t_4), (o_5, t_5) \) (Recall Figure 2 for its representation).

**B. Evolutionary Operators**

In this section detailed discussions of random chromosome generation as well as the other four phases of the GA are provided.

1) **Random Chromosome Generation:** Notice that generating chromosomes at random can not be accomplished by solely picking a random sequence of atomic objectives, since each chromosome must be a trace of the given specification. In this section, we provide an algorithm, which randomly generates a chromosome, i.e., a trace of the specification, such that there is a nonzero probability for any trace of the specification to be chosen. The algorithm heavily employs a procedure denoted as \texttt{Next}, which maps a given term \( p \) to the set of all pairs \((p', o')\) of terms and atomic objectives such that for any \((p', o')\) in \texttt{Next}(\( p \)) we have that \( p \) can evolve into \( p' \) by executing \( o' \), i.e., the transition \( p \rightarrow_{o'} p' \) holds. An algorithmic procedure to compute \texttt{Next}(\( p \)) is provided in Algorithm 1, the correctness of which follows easily from the operational semantics of process algebra.

The \texttt{Next} algorithm runs recursively. Its execution is best visualized with the parse tree of the process algebra term, \( p \), that it takes as a parameter. By the semantics of alternative composition operator, the atomic objectives that can be executed next by \( p = p_1 + p_2 \) is exactly those that can be executed either by \( p_1 \) or \( p_2 \). Hence, if the root node of the parse tree of \( p \) is a + operator, i.e., \( p \) is of the form \( p = p_1 + p_2 \), then the algorithm calls itself recursively with parameters \( p_1 \) and \( p_2 \), and returns the union of that returned after these two calls (Lines 2-3). The semantics of the sequential composition operator is such that the atomic objectives that can be executed next by \( p = p_1 \cdot p_2 \) is exactly those that can be executed by \( p_1 \); However, if \( p_1 \) evolves to \( p_1' \) after executing an atomic objective, \( p \) evolves to \( p_1' \cdot p_2 \) after executing the same atomic objective. Hence, if \( p \) is of the form \( p = p_1 \cdot p_2 \), then the algorithm calls itself recursively with parameter \( p_1 \), concatenates \( p_2 \) to the end of \( p_1' \) in all pairs \((p_1', t') \) returned by this recursive call, and returns all the resulting pairs (Lines 4-5). The semantics of the parallel composition operator is such that \( p = p_1 \parallel p_2 \) can execute the atomic objectives that either \( p_1 \) or \( p_2 \) can execute. If, \( p \) is of the form \( p_1 \parallel p_2 \), then the algorithm recursively calls itself twice with parameters \( p_1 \) and \( p_2 \). The pairs returned after these calls are appropriately concatenated by \( p_1 \) and \( p_2 \), and all resulting pairs are returned (Lines 6-7). Finally, if the \( p = o \), where \( o \) is an atomic objective, then the algorithm returns \((\sqrt{o}, o) \) (Lines 8-9), since \( p \) can only execute the atomic objective \( o \) and evolve to the terminated process, \( \sqrt{o} \).

**Algorithm 1: \texttt{Next}(p) Procedure**

1. switch \( p \) do
2. \quad \text{case} \( p = p_1 + p_2 \) \begin{align*}
3. & \quad \text{return} \texttt{Next}(p_1) \cup \texttt{Next}(p_2) \\
4. & \quad \text{case} \ p = p_1 \cdot p_2 \begin{align*}
5. & \quad \text{return} \{(p_1', p_2, t') | (p_1', t') = \texttt{Next}(p_2)\} \\
6. & \quad \text{case} \ p = p_1 \parallel p_2 \begin{align*}
7. & \quad \text{return} \{(p_1' \parallel p_2, t') | (p_1', t') = \texttt{Next}(p_1)\} \cup \\
8. & \quad \{(p_1 \parallel p_2', t') | (p_2', t') = \texttt{Next}(p_2)\} \\
9. & \quad \text{case} \ p = o \in \mathcal{O} \begin{align*}
10. & \quad \text{return} \{(\sqrt{o}, o)\}
11. \end{align*}
12. \end{align*}
\end{align*}
13. \text{endsw}

Algorithm 1 recursively explores the parse tree of \( p \) in this manner, and extracts all the atomic objectives that can be executed next. Note that each node in the parse tree is explored at most once in the algorithm, which implies that the running time of the algorithm is linear with respect to the size of the specification even in the worst case.

Given a finite set \( S \) of elements, let \texttt{Rand}(\( S \)) be a procedure that returns an element of \( S \) uniformly at random. The algorithm that generates a random chromosome, denoted as \texttt{RandomGenerate}, is given in Algorithm 2. The \texttt{RandomGenerate}(\( p \)) procedure runs the \texttt{Next}(\( p \)) procedure...
(Line 4), randomly picks one of the atomic objectives returned by Next, say \( o' \) (Line 5), and runs itself recursively with the process \( p' \) that \( p \) evolves to after executing \( o' \) (Line 6). The recursion ends when the algorithm is run with the terminated process (Line 2). The chromosome that is returned is essentially a concatenation of the atomic objectives that were picked randomly along the way during the recursion.

Algorithm 2: RandomGenerate\( (p) \) Procedure

\[
\begin{align*}
\text{if } p &= \sqrt{\text{Next}(p)} \\
\text{return } o \\
\text{else} \\
S &\leftarrow \text{Next}(p) \\
(p', o') &\leftarrow \text{Random}(S) \\
\gamma' &\leftarrow \text{RandomGenerate}(p') \\
\text{return } o' \mid \gamma'
\end{align*}
\]

Note that Algorithm 2 returns exactly one random trace of \( p \). Using Algorithm 2 repeatedly, however, a set \( X \) of chromosomes can be generated to initialize the GA. After this initialization, each iteration of the GA proceeds with the aforementioned five phases, detailed after the next short example.

Example To illustrate the random chromosome generation, consider the running example specification \( p_{\text{spec}} = (o_1 + o_2) \cdot (o_3 || (o_4 \cdot o_5)) \).

The algorithm RandomGenerate\( (p_{\text{spec}}) \) first calls the Next procedure with \( p_{\text{spec}} \), which returns \( S = \{ (p_1, o_1), (p_2, o_2) \} \), where \( p_1 = p_2 = o_3 \parallel (o_4 \cdot o_5) \). Say the Random\( (S) \) procedure returns the pair \( (p_2, o_2) \) among the two elements of \( S \). Then, the RandomGenerate algorithm calls itself with \( p_2 \), which generates a second call to Next procedure, this time with \( p_2 \), which returns \( \{ (p_3, o_3), (p_4, o_4) \} \), where \( p_3 = o_1 \cdot o_5 \) and \( p_4 = o_4 \parallel o_5 \). Assume, that in this iteration, the Rand\( (S) \) procedure returns \( (p_4, o_1) \). The RandomGenerate algorithm, this time, runs itself with \( p_4 \). The Next\( (p_4) \) call returns \( S = \{ (p_5, o_3), (p_6, o_5) \} \), where \( p_5 = o_3 \) and \( p_6 = o_5 \). Assume that Rand\( (S) \) returns \( (p_5, o_3) \). Finally, the RandomGenerate procedure calls itself with \( p_5 \), which results in Next\( (p_5) \) returning \( S = \{ \sqrt{\text{o}_3} \} \) and Rand\( (S) \) returning \( o_3 \). The final call of RandomGenerate with \( \sqrt{\text{o}_3} \), then, returns \( o_3 \) and the procedure terminates. After termination the random chromosome obtained is \( X = (o_2, o_4, o_3, o_5) \).

2) Selection: In the selection phase, pairs of chromosomes are selected randomly from the set \( X \) of all chromosomes to be the parents of the next generation. The randomization is biased, however, so that those chromosomes that are more “fit” than others are selected to be parents with higher probability. Throughout this paper, the fitness of a chromosome is evaluated using the cost of its corresponding schedule as follows:

\[ f_{X_i} = \frac{1}{J_i(S(X_i))} \]

where \( i = 1, 2 \) (see Equation (1)). That is, the chromosomes with lower-cost corresponding schedules are rated as more fit ones.

After the selection phase, a child chromosome is produced from these two parent chromosomes via the crossover operation.

3) Crossover: The crossover operation generates a child chromosome \( X' \) from a given pair \( X_1 \) and \( X_2 \) of parent chromosomes. Note that merely picking a cutting point and joining parts of two valid chromosomes does not necessarily produce a valid chromosome in this case. In this section, a cut and splice crossover operator that always produces a valid chromosome is provided.

Informally speaking, the crossover operation first partitions the set \( O \) of atomic objectives into two sets of atomic objectives denoted as \( S_1 \) and \( S_2 \). Then, two different sequences, \( \sigma_1 \) and \( \sigma_2 \), are formed such that \( \sigma_i \) is the order preserving projection of \( X_i \) onto the set \( S_i \) for \( i = 1, 2 \). In the end, the child chromosome is the concatenation of \( \sigma_1 \) and \( \sigma_2 \).

Let us first identify four primitive procedures, which help clarify the presentation of the crossover algorithm. Let \( p \) be a process algebra term. The procedure \( \text{ChildrenAO}_p(n) \) takes a node \( n \) of the parse tree of \( p \) and returns the set of all actions (equivalently, atomic objectives) that are labels of the leaf nodes of the tree rooted at \( n \). An algorithmic procedure for computing \( \text{ChildrenAO}_p(n) \) is given in a recursive form in Algorithm 3. The procedure RightMostChild\( (n) \) returns the rightmost leaf of the tree rooted by node \( n \). This function is presented in an algorithmic form in Algorithm 4. Given an atomic objective \( o \in O \), let the functions Left\( (o) \) and Right\( (o) \) return the set of atomic objectives that are, intuitively speaking, to the left of \( o \) and right of \( o \), respectively. More precisely, we have \( \sigma_1 \in \text{Left}(o) \) and \( \sigma_2 \in \text{Right}(o) \) if and only if there exists \( n, m_{\text{left}}, m_{\text{right}} \in N_p \) such that:

- \( (m_{\text{left}}, m_{\text{right}}) = \text{ChildrenAO}(n) \),
- \( o \in \text{ChildrenAO}_{m_{\text{left}}}(n) \) but \( o \notin \text{ChildrenAO}_{m_{\text{right}}}(n) \),
- or \( o \notin \text{ChildrenAO}_{m_{\text{right}}}(m_{\text{left}}) \) but \( o \notin \text{ChildrenAO}_{m_{\text{left}}}(m_{\text{right}}) \).

The procedure Right\( p(o) \) is defined symmetrically.

As mentioned earlier, the crossover algorithm first creates two disjoint sets \( S_1 \) and \( S_2 \) of atomic objectives, such that \( S_1 \cup S_2 = O \). The sets \( S_1 \) and \( S_2 \) are, indeed, formed using a natural ordering of the atomic objectives, which comes from the parse tree itself. Intuitively speaking, the atomic objectives in a parse tree can be ordered such that \( o_1 \prec_p o_2 \) if and only if \( o_1 \) is to the left of \( o_2 \) in the tree.

More precisely, first a “cutting point” atomic objective, say \( o \), is chosen according to some procedure to be outlined shortly, and \( S_1 \) and \( S_2 \) are defined as \( S_1 := \text{Left}(o) \cup \{ o \} \) and \( S_2 := \text{Right}(o) \). It can be shown rather easily that this selection of \( S_1 \) and \( S_2 \) satisfies \( S_1 \cap S_2 = \emptyset \) and \( S_1 \cup S_2 = O \). Then, the child chromosome can be generated using \( S_1 \) and \( S_2 \) as outlined above. Note, however, that not all choices of \( o \) would yield a valid chromosome, i.e., a trace of the specification. Yet, it is possible to select the cutting point atomic objective so that the resulting chromosome will be valid. Such a procedure runs as follows. Informally speaking, first, the parse tree of the specification is randomly rearranged, while preserving the behavior (set of traces that can be generated) of the specification. The rearrangement is done by randomly choosing to either switch the left and right children...
of each alternative and parallel composition operator in the parse tree or keep them as is. This procedure is denoted as RandomRearrange, which takes a process \( p \) and returns the rearranged one. RandomRearrange allows a variety of different schemes for cutting the parent chromosomes, as will be clear later. Then, an atomic objective \( o_{\text{rand}} \) is selected at random among \( O \), and used as a cutting point if the resulting child chromosome yields a feasible assignment. If not, the “nearest” atomic proposition, to the right of \( o_{\text{rand}} \), that would yield a feasible assignment. This procedure is given in Algorithm 5.

Algorithm 5: CutAtomicObjective\(_p\)(\(X_1, X_2\)) Procedure

1. \( o_{\text{rand}} \leftarrow \text{Rand}(O) \)
2. \( S \leftarrow \{ o_{\text{rand}} \} \)
3. while \( (o \notin X_1 \text{ for } \forall o \in S) \text{ or } (o \notin X_2 \text{ for } \forall o \in S) \) do
   4. \( \text{while } \text{Operator}_p(n) = + \) do
      5. \( n \leftarrow \text{Parent}_p(n) \)
   6. \( S \leftarrow \text{ChildrenAO}_p(n) \)
7. end
8. if \( \text{Leaf}_p(n) = \text{False} \) then
   9. \( n \leftarrow \text{RightMostChild}_p(n) \)
10. end
11. \( o_{\text{cut}} \leftarrow \text{Action}_p(n) \)
12. return \( o_{\text{cut}} \)

The crossover operation is summarized in Algorithm 6. Given two chromosomes \( X_1 \) and \( X_2 \), the crossover algorithm first generates a cut point atomic objective \( o_{\text{cut}} \) via Algorithm 5. In the second step, it generates the two sets \( S_1 := \text{Left}(o_{\text{cut}}) \cup \{ o_{\text{cut}} \} \) and \( S_2 := \text{Right}(o_{\text{cut}}) \), which represent the set of atomic objectives to the left of \( o_{\text{cut}} \) and the ones to the right of \( o_{\text{cut}} \), respectively. Using these two sets, two sequences, \( \sigma_1 \) and \( \sigma_2 \), are generated from chromosomes \( X_1 \) and \( X_2 \). Finally, the resulting child chromosome \( X' \) is the concatenation of the two sequences \( \sigma_1 \) and \( \sigma_2 \).

Algorithm 6: Crossover\(_p\)(\(X_1, X_2\)) Procedure

1. \( p' \leftarrow \text{RandomRearrange}(p) \)
2. \( o_{\text{cut}} \leftarrow \text{CutAtomicObjective}_p(X_1, X_2) \)
3. \( S_1 := \text{Left}_p(o_{\text{cut}}) \cup \{ o_{\text{cut}} \} \)
4. \( S_2 := \text{Right}_p(o_{\text{cut}}) \)
5. \( \sigma_1 := [X_1]_{S_1} \)
6. \( \sigma_2 := [X_2]_{S_2} \)
7. return \( \sigma_1 \sigma_2 \)

Example Consider the running example with the specification \( p_{\text{spec}} = (o_1 + o_2) \cdot (o_3 | (o_4 + o_5)). \) Consider the two chromosomes \( X_1 = (o_1, o_3, o_4, o_5) \) and \( X_2 = (o_2, o_4, o_3, o_5) \). First, the crossover operation calls the random rearrange procedure, which switches the left and right children of alternative and parallel composition operators or keeps them as is with equal probability. The parse tree of \( p_{\text{spec}} \) was given in Figure 1. Notice that there is exactly one alternative and one parallel operator, each of which can have their children switched with probability 1/2. Let us assume that the RandomRearrange procedure switches the children of the parallel composition operator, while keeping unchanged that of the alternative composition operator. The new parse tree is shown in Figure 4. Next, the CutAtomicObjective\(_p\) procedure is run with \( X_1 \) and \( X_2 \). Assume that the \( o_{\text{rand}} \) turns out to be \( o_4 \), which is included in both of the chromosomes. Hence, CutAtomicObjective\(_p\) procedure returns \( o_{\text{cut}} = o_4 \). From the parse tree presented in Figure 4, notice that we have \( S_1 = \text{Left}_p(o_{\text{cut}}) \cup \{ o_{\text{cut}} \} = \{ o_1, o_2, o_4 \} \) and \( S_2 = \text{Right}_p(o_{\text{cut}}) = \{ o_3, o_5 \} \). Hence, we get \( \sigma_1 = [X_1]_{S_1} = (o_1, o_4) \) and \( \sigma_2 = [X_2]_{S_2} = (o_3, o_5) \). Thus, we find that \( X = (o_1, o_3, o_4, o_5), \) which is also a trace of \( p_{\text{spec}}, \) thus a valid chromosome.

Notice that this crossover procedure combined the characteristics of the parent chromosomes, \( X_1 \) and \( X_2 \), in the child chromosome \( X \). The choice of executing \( o_1 \) rather than \( o_2 \) is inherited from \( X_1 \), whereas the choice of executing \( o_3 \) after \( o_4 \) is inherited from \( X_2 \).
Consider the chromosome algorithm on this example.

**Procedure (Algorithm 2):**
- `RandomGenerate` operator can be represented in two different ways. Consider, $X$ chromosomes are picked at random to another chromosome and carried over to the next generation. However, this operation, again, is not trivial, since the modified chromosome must also be a valid chromosome, i.e., a trace of the specification $p_{\text{spec}}$.

The mutation phase, given in Algorithm 7, proceeds as follows. Firstly, a number of chromosomes are picked at random from the set $X$ of chromosomes. Secondly, for each such chromosome $X$, an atomic objective $o_{\text{mid}} \in X$ is picked uniformly at random. Then, $X$ is partitioned into two sequences, $\sigma_1$ and $\sigma_2$, such that $\sigma_1^i \sigma_2^i = X$ and $\sigma_1^i(1) = o_{\text{mid}}$. Finally, the mutated chromosome $X^\prime$ is computed by employing the RandomGenerate procedure (Algorithm 2): $X^\prime = \sigma_1^\prime o_{\text{rand}} \sigma_2^\prime$, where $o_{\text{rand}} = \text{RandomGenerate}(p')$ and $p' \in \mathbb{T}$ is such that $p \xrightarrow{\sigma} p'$.

**Algorithm 7: Mutate$_p(X)$ Procedure**

1. $o_{\text{mid}} \leftarrow \text{Rand}(\{ o \mid o \in X \})$
2. $\sigma_1, \sigma_2 \in \Sigma_2$ are such that $p = \sigma_1^i \sigma_2^i$ and $\sigma_1^i(1) = o_{\text{mid}}$
3. $p'$ is such that $p \xrightarrow{\sigma} p'$
4. $o_{\text{rand}} \leftarrow \text{RandomGenerate}(p')$
5. return $\sigma_1^i o_{\text{rand}} \sigma_2^i$

In essence, after the chromosome is partitioned into two, the first part of the chromosome is kept whereas the second part is re-generated randomly.

**Example** Consider the chromosome $X = (o_2, o_4, o_3, o_5)$ from the previous example. Let us illustrate the mutation algorithm on this example.

The mutation operator first picks an element of $X$ using a uniformly random distribution. Let us assume that this element turns out to be $o_3$. Then, the algorithm partitions the chromosome into two as in $X = \sigma_1 \sigma_2$, where $\sigma_1 = (o_2, o_4)$ and $\sigma_2 = (o_3, o_5)$. Notice that $p \xrightarrow{\sigma} p'$ holds, where $p' = (o_2, o_4, o_5)$. The mutation operator, then, runs the RandomGenerate procedure with $p'$, which might either return $\sigma_3 = (o_3, o_5)$ or $\sigma_4 = (o_5, o_3)$. Assume that it returns the latter. Then, the resulting mutated chromosome is the concatenation of $\sigma_1$ and $\sigma_4$, i.e., $(o_2, o_4, o_5, o_3)$.

**5) Elitism:** In the elitism phase, the chromosomes with high fitness are selected to move into the next generation. The selection is made randomly, even though biased towards chromosomes with high fitness values. However, we deterministically choose a small set of chromosomes, called the elite members, of the current generation with the highest fitness values, in order to rule out the possibility of loosing all the good solutions in a given generation. This provides us with a solution that is monotonically improving.

**C. Algorithm and Correctness**

Let us extend the primitive procedure Rand as follows. Let $\text{Rand}(S, \phi, k)$ be a primitive procedure, where $S$ is a finite set, $\phi : S \rightarrow \mathbb{R}_+$ is a function, and $k$ is number such that $k < |S|$. The function $\text{Rand}(S, \text{value}, k)$ returns a set of $k$ distinct elements from $S$ by randomly picking an element $s \in S$ with probability $\phi(s)/\sum_{s' \in S} \phi(s')$ repeatedly, until $k$ distinct elements are selected. Let also $\text{SelectBest}(S, \phi, k)$ be another procedure that returns the $k$ elements with highest values of the function $\phi$. More precisely, $\text{SelectBest}(S, \phi, k)$ is a set of $k$ elements from set $S$ such that for all $s \in \text{SelectBest}(S, \phi, k)$ and for all $s' \in S \setminus \text{SelectBest}(S, \phi, k)$ we have that $\phi(s) \geq \phi(s')$.

The GA is formalized in Algorithm 8, where the initialization phase (Lines 2-4) as well as the selection (Lines 6-8), crossover (Lines 9-16), mutation (Lines 17-23), and elitism (Line 24) operations are shown explicitly.

**Algorithm 8: GA($p_{\text{spec}}, K_{\text{total}}, K_{\text{children}}, K_{\text{elite}}, K_{\text{mutate}}, N$)**

1. $X \leftarrow \emptyset$
2. for $i \leftarrow 1$ to $K_{\text{total}}$ do
3. $X \leftarrow X \cup \text{RandomGenerate}(p_{\text{spec}})$
4. end
5. for $j \leftarrow 1$ to $N$ do
6. for all $X_k \in X$ do
7. fitness($X_k$) $\leftarrow 1/J(\text{PC}(X_k))$
8. end
9. $C \leftarrow \emptyset$
10. for $i \leftarrow 1$ to $K_{\text{children}}$ do
11. $X_1 \leftarrow \text{Rand}(X, \text{fitness}, 1)$
12. $X_2 \leftarrow \text{Rand}(X, \text{fitness}, 1)$
13. $C \leftarrow \text{Crossover}(X_1, X_2)$
14. fitness($C$) $\leftarrow 1/J(\text{PC}(C))$
15. $C \leftarrow C \cup C$
16. end
17. $M \leftarrow \emptyset$
18. for $i \leftarrow 1$ to $K_{\text{mutate}}$ do
19. $X \leftarrow \text{Rand}(X, \text{fitness}, 1)$
20. $M \leftarrow \text{Mutate}(X)$
21. fitness($M$) $\leftarrow 1/J(\text{PC}(C))$
22. $M \leftarrow M \cup M$
23. end
24. $X \leftarrow \text{SelectBest}(X, K_{\text{elite}}) \cup \text{Rand}(X \cup C \cup M, \text{fitness}, K_{\text{total}} - K_{\text{elite}})$
25. end
26. return SelectBest($X, 1$)
Let us discuss the correctness of the algorithm. Firstly, notice that the RandomGenerate($p_{spec}$) function presented in Algorithm 2 is correct in the sense that it returns only those traces that are in $\Gamma_{p_{spec}}$. This fact follows from the correctness of the Next function, for which an algorithmic procedure was presented in Algorithm 1. Let us show that each trace in $\Gamma_{p_{spec}}$ is selected by RandomGenerate($p_{spec}$) with non-zero probability.

**Proposition V.1** Given any specification $p_{spec}$ $\in T$ defined on a given set of atomic objectives $O$, the probability that $\gamma =$ RandomGenerate($p_{spec}$) is at least $1/|O|^{|O|}$.

This proposition guarantees that the first generation has the set of all chromosomes as its support. It also allows the mutation operator to consider all possible options during mutation.

Next, let us note that the crossover operation is correct in the sense that Crossover($X_1, X_2$) returns a valid chromosome whenever $X_1$ and $X_2$ are valid chromosomes.

**Proposition V.2** Given any specification $p_{spec}$ $\in T$ and any two chromosomes $X_1$ and $X_2$ such that $X_1, X_2 \in \Gamma_{p_{spec}}$, then we have that Crossover($X_1, X_2$) $\in \Gamma_{p_{spec}}$.

VI. SIMULATIONS

The algorithm described in this paper was implemented in the C++ programming language. This section is devoted to a simulation study evaluating the effectiveness of the algorithms in small-, medium-, and large-scale examples. All the simulations were run on a laptop computer equipped with a 2.66 GHz processor and 4 GB RAM running the Linux operating system.

First, we consider a simple scenario where 2 UAVs are required to engage 5 targets. Each target has to be first classified and then serviced, and all the targets can be serviced independently, i.e., in parallel. This problem instance can be described within the process algebra framework described in this paper using 20 atomic objectives. Let $o_{i,c,u}$ be the atomic objective that indicates classification (subscript $c$ is used for classification and $s$ is used for servicing) of the target $i \in \{1, 2, 3, 4, 5\}$ by UAV $u \in \{1, 2\}$. The generic objective of servicing target $i$ can be written as $o_i = (o_{i,c,1} + o_{i,s,1}) \cdot (o_{i,c,2} + o_{i,s,2})$. Then the mission specification is $p_{spec} = o_1 \parallel o_2 \parallel o_3 \parallel o_4 \parallel o_5$. In all the examples presented in this section, we use $J_2$ (see Equation 1) as our cost metric.

This example scenario is small enough that our C++ implementation of the tree search algorithm given in Ref. [7] terminates in about a minute with the optimal solution, which has cost 1.4 in this case. To evaluate the genetic algorithm presented in this paper, we have done a Monte Carlo simulation study. We have considered three different parameter sets (see Table I) and for each of the parameter sets we have run the algorithm 100 times. The random parameters in this study were those associated with the implementation of the GA. The average cost vs. the average running time is plotted in Figure 5 for all the three parameter sets. Notice that for all the parameter sets, in average, the genetic algorithm gets very close to the optimal solution (of 1.4) very quickly. In fact, in most runs the algorithm achieves the optimal solution in a few seconds. In Figure 6, the percentage of trials, for which the solution is the optimal and within 1%, 5%, 10%, and 15% of the optimum is plotted for all the three parameter sets. Notice that in less than a second in almost all the trials the algorithm finds a solution that is within 15% percent of the optimum, no matter what parameter set is used. Notice that, for parameter set 3, within the first two seconds, in all trials the solution is within 10% of the optimum, in 98% of the trials the solution is within 5% of the optimum, and at the end of 10 seconds of computation time in 80% of the trials the solution is within 1% of the optimum, and in 60% of the trials the solution is the optimum solution itself. Notice also that in this example with 20 atomic objectives parameter set 3, i.e., bigger population, seems to converge to the optimum solution more quickly. However, this gap between different parameter sets seems to diminish in larger-scale examples.

To evaluate the genetic algorithm in medium-scale examples, we consider the same scenario, this time with 3 identical UAVs, which induces an example with 30 atomic objectives. This scenario is large enough that our implementation of the three search algorithm runs out of memory before termination. The performance of the genetic algorithm is shown in Figure 7 for all three different parameter sets and the closeness of the solutions to the best solution found in any trial is shown in Figure 8 for the third parameter set only. In an even larger-scale example, we consider 4 identical UAVs in the same scenario, which can be modeled by 40 atomic objectives. The results are shown in Figures 9 and 10. Notice that all the parameter sets perform similarly in these medium-scale scenarios. Comparing Figures 7 and 9, clearly, the cost of the solution returned by the algorithm decreases with increasing number of UAVs employed in the mission, since the cost function is the time that the mission is completed.

Next we fix the number of UAVs to 3 and increase the num-

![Monte-Carlo simulation results for a small-scale scenario involving 5 targets and 2 UAVs. Cost of the best chromosome in a generation is averaged over trials and plotted against average running time for all the three parameter sets.](attachment:monte-carlo_simulation.png)

**Table I**

<table>
<thead>
<tr>
<th>Parameter Set</th>
<th>$K_{total}$</th>
<th>$K_{elite}$</th>
<th>$K_{mutate}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>100</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>1000</td>
<td>200</td>
<td>200</td>
</tr>
</tbody>
</table>

PARAMETER SETS.
Fig. 6. Monte-Carlo simulation results for a small-scale scenario involving 5 targets and 2 UAVs. The percentage of trials that achieve the optimal solution, as well as those that are within 1%, 5%, 10%, and 15% of the optimal are plotted against the average running time of the algorithm, for parameter sets 1, 2, and 3 in Figures (a), (b), and (c), respectively.

Fig. 7. Monte-Carlo simulation results for a medium-scale scenario involving 5 targets and 3 UAVs. Cost of the best chromosome in a generation is averaged over trials and plotted against average running time for all the three parameter sets in the simulation example with 5 targets and 3 UAVs. The figure is in semi-logarithmic scale.

Fig. 8. Monte-Carlo simulation results for a medium-scale scenario involving 5 targets and 3 UAVs. The percentage of trials that achieve the best solution found in all the trials as well as those that are within 10% and 15% of the best solution are plotted against the average running time of the algorithm for parameter set 3 (the other two parameter sets produce similar solutions for this problem instance).

Fig. 9. Monte-Carlo simulation results for a medium-scale scenario involving 5 targets and 4 UAVs. Cost of the best chromosome in a generation is averaged over trials and plotted against average running time for all the three parameter sets in the simulation example with 5 targets and 4 UAVs. The figure is in semi-logarithmic scale.

Fig. 10. Monte-Carlo simulation results for a medium-scale scenario involving 5 targets and 4 UAVs. The percentage of trials that achieve the best solution found in all the trials as well as those that are within 10% and 15% of the best solution are plotted against the average running time of the algorithm for parameter set 3 (the other two parameter sets produce similar solutions for this problem instance).

VII. CONCLUSIONS

A genetic algorithm for planning based on process algebra was presented in this paper. The applicability of the approach was demonstrated on an example complex mission planning problem involving cooperative UAVs. It was shown that process algebra can be used to formally define the evolutionary operators of crossover and mutation. The viability of the approach was investigated in Monte-Carlo simulations of small-, medium-, and large-scale problems. It was shown that, for a small sized problem, on the average the algorithm converges...
to the optimal solution in a few seconds. On larger-scale scenarios, in which optimal algorithms such as the tree search can not be used, the GA was shown to quickly produce a good solution and improve it to better solutions within the first minute.

Although the paper mostly concentrates on a vehicle routing setting, the algorithms presented in this paper can be applied to a variety of combinatorial optimization problems, where the combinatorial complexity in the problem can be described naturally using the process algebra framework. Future work includes identification of such problems and the application of this framework in a broader domain. In fact, the effectiveness of stochastic search combined with computationally efficient process algebra specifications may prove to be useful in many optimization problems in engineering.

This paper concentrated only on missions which come to an end after the specification is successfully fulfilled. Specification and planning of contingent persistent missions, i.e., missions which have to continue forever while considering adversarial actions, such as surveillance, is interesting on its own right. Another direction for future work includes a formal study of such problems to design effective algorithms that can also take contingencies into account.

**ACKNOWLEDGMENTS**

This research was partially supported by the Michigan/AFRL Collaborative Center on Control Sciences, AFOSR grant no. FA 8650-07-2-3744; and by AFOSR, Air Force Material Command, grant no. FA8655-09-1-3066.

**REFERENCES**

such that

\[ \{ \sigma_1 | \sigma_2 : \sigma_1 \in \Gamma_{p_1}, \sigma_2 \in \Gamma_{p_2} \} \]

Note that the RandomRearrange operator does not exchange the children of the root node. Thus, we have that

\[ \Gamma_{p'} = \{ \sigma_1 | \sigma_2 : \sigma_1 \in \Gamma_{p_1}, \sigma_2 \in \Gamma_{p_2} \} \]

which is equal to \( \Gamma_p \).

Finally, if the root node of the parse tree is a parallel composition operator, then we have that

\[ \Gamma_p = \bigcup_{\sigma_1 \in \Gamma_{p_1}, \sigma_2 \in \Gamma_{p_2}} \mathcal{C}(\sigma_1, \sigma_2) \]

where \( \mathcal{C}(\sigma_1, \sigma_2) \) is the set of concurrent compositions of \( \sigma_1 \) and \( \sigma_2 \) defined so that \( \sigma \in \mathcal{C}(\sigma_1, \sigma_2) \) whenever the order preserving projections \( [\sigma]_1 \) and \( [\sigma]_2 \) equal to \( \sigma_1 \) and \( \sigma_2 \), respectively, where \( S_i \) is the set of all atomic objectives that appear in \( \sigma_i \) for \( i = 1, 2 \).

Notice that if RandomRearrange does not exchange the left and right children of the root node of the parse tree, we have that

\[ \Gamma_{p'} = \bigcup_{\sigma_1 \in \Gamma_{p_1}, \sigma_2 \in \Gamma_{p_2}} \mathcal{C}(\sigma_1, \sigma_2) \]

In the former case, we have that \( \Gamma_{p'} = \Gamma_p \) since we have \( \Gamma_{p_i} = \Gamma_{p_i} \) for \( i = 1, 2 \) by the induction hypothesis. In the latter case, however, the situation is identical to the former case since \( \mathcal{C}(\sigma_1, \sigma_2) = \mathcal{C}(\sigma_2, \sigma_1) \).

Hence, in either case, the hypothesis is satisfied.

Let us prove the proposition by induction on the depth of the parse tree. For the base case when the depth is one, the hypothesis trivially holds, since such a process \( p \) is of the form \( p = o \), where \( o \in O \). Hence, \( X_1 = X_2 = (o) \), since there is only one valid chromosome. Notice that in this case \( \text{Crossover}_p(X_1, X_2) \) also returns \( (o) \), which is a valid chromosome. Assume that the hypothesis holds whenever the parse tree of \( p \) has depth \( n \). To show that it holds for parse trees of depth \( n+1 \), let us consider three cases: \( p \) is of the form \( p_1 + p_2, p_1 | p_2 \), or \( p_1 \parallel p_2 \), i.e., the root of the parse tree encodes an alternative, sequential, or parallel composition operator.

In the first case when \( p = p_1 + p_2 \), notice that each parent chromosome is either a trace of \( p_1 \) or one of \( p_2 \). In the case when both parents are traces of the same process, say \( p_1 \), and \( o_{\text{rand}} \in \text{ChildrenAO}(p_1) \) (Lines 3-8 of Algorithm 5 are not executed), then the resulting child chromosome is merely the outcome of \( X = \text{Crossover}_p(X_1, X_2) \). However, since \( p_1 \) has depth \( n \), \( X \) is a valid chromosome in this case by the induction hypothesis and by the fact that \( \Gamma_{p_1} = \Gamma_{p_1} \). If, on the other hand, \( o_{\text{rand}} \notin \text{ChildrenAO}(p_1) \), then we have that \( o_{\text{cut}} \in \text{ChildrenAO}(p_2) \). In this case, \( o_{\text{cut}} \) returned by the CutAtomicObjective procedure is the atomic proposition encoded by the rightmost leaf node in the parse tree of \( p_2 \).

Thus, we have \( S_1 = O \) and \( S_2 = \emptyset \), which yields \( X = X_1 \). Hence, \( X \) is a valid chromosome in this case also. The case when both parents are traces of \( p_2 \) is symmetric. In the case when \( X_1 \) is a trace of \( p_1 \) and \( X_2 \) is a trace of \( p_2 \), notice that \( o_{\text{cut}} \) is the atomic proposition encoded by the rightmost leaf node of parse tree of \( p_2 \).

Finally, the case when \( X_1 \) is a trace of \( p_2 \) and \( X_2 \) is a trace of \( p_1 \) is symmetric.

For the case when \( p = p_1 \cdot p_2 \), both parents \( X_1 \) and \( X_2 \) are of the form \( X_1 = \sigma_{1,1} | \sigma_{1,2} \) and \( X_2 = \sigma_{2,1} \sigma_{2,2} \), where \( \sigma_{1,1}, \sigma_{1,2} \in \Gamma_{p_1} \) and \( \sigma_{2,1}, \sigma_{2,2} \in \Gamma_{p_2} \). If \( o_{\text{rand}} \in \text{ChildrenAO}(p_1) \), then we have that \( X = \sigma_{1,1} | \sigma_{1,2} \), where \( \sigma_2 = X_2 \) (thus \( \sigma_2 \) is a trace of \( p_2 \)) and \( \sigma_1 = \sigma_{1,1} | \sigma_{1,2} \).
Crossover, \( p_1(\sigma_1, \sigma_2) \), which returns a valid chromosome, i.e., a trace of \( p_1' \), which is also a trace of \( p_1 \) by Lemma B.1. Hence, \( \sigma_1|\sigma \) is a trace of \( p_1 \cdot p_2 \). The case when \( o_{\text{rand}} \in \text{ChildrenAO}(p_2) \) is symmetric. The case when \( p = p_1 \parallel p_2 \) is very similar to the previous case and is omitted here. \( \square \)