Constraints Enabled Autonomous Agent Marketplace: Discovery and Matchmaking

Debmalya Biswas
Wipro AI, Switzerland

Keywords: Multi-Agent Systems, Autonomous Agents, Discovery, Constraints, Composition, Marketplace.

Abstract: The recent advances in Generative AI have renewed the discussion around Auto-GPT, a form of autonomous agent that can execute complex tasks, e.g., make a sale, plan a trip, etc. We focus on the discovery aspect of agents, i.e., identifying the agent(s) capable of executing a given task. This implies that there exists a marketplace with a registry of agents - with a well-defined description of the agent capabilities and constraints. In this paper, we outline a constraints based model to specify agent services. We show how the constraints of a composite agent can be derived and described in a manner consistent with respect to the constraints of its component agents. Finally, we discuss approximate matchmaking, and show how the notion of bounded inconsistency can be exploited to discover agents more efficiently.

1 INTRODUCTION

In the Generative AI context, Auto-GPT (aut, 2023) is representative of an Autonomous AI Agent that can execute complex tasks, e.g., make a sale, plan a trip, make a flight booking, book a contractor to do a house job, order a pizza. Given a user task, Auto-GPT aims to identify (compose) an agent (group of agents) capable to executing the given task. A high-level approach to solving such complex tasks involves: (a) decomposition of the given complex task into (a hierarchy or workflow of) simple tasks, followed by (b) composition of agents able to execute the simpler tasks. This can be achieved in a dynamic or static manner. In the dynamic approach, given a complex user task, the system comes up with a plan to fulfill the request depending on the capabilities of available agents at run-time. In the static approach, given a set of agents, composite agents are defined manually at design-time combining their capabilities.

Generative agents (Park et al., 2023) follow a long history of research around Multiagent Systems (MAS) (Weiss, 2016), especially, Goal Oriented Agents (Bordes et al., 2017; Yan et al., 2015). The main focus of this paper is on the discovery aspect of agents, i.e., identifying the agent(s) capable of executing a given task. This implies that there exists a marketplace (aim, 2023) with a registry of agents, with a well-defined description of the agent capabilities and constraints.

For example, let us consider a House Painting Agent C whose services can be reserved online (via credit card). Given this, the fact that the user requires a valid credit card is a constraint and the fact that the user's house will be painted within a certain time frame are its capabilities. In addition, we also need to consider any constraints of C during the actual execution phase, e.g., the fact that C can only provide the service on weekdays. The above restriction might be a problem if the user specified task requires getting the work done during weekends. In general, constraints refer to the conditions that need to be satisfied to initiate an execution and capabilities reflect the expected outcome after the execution terminates.

We have seen similar efforts in the past with API marketplaces (API, 2023) and Universal Description, Discovery and Integration (UDDI) (UDD, 2000) registries in the context of Web Services (Services Oriented Computing). However, the service descriptions captured in XML or JSON are too static, and lack the necessary semantic information for a capabilities "negotiation" during their discovery.

In the context of MAS, specifically, previous works (Capezzuto et al., 2021; Trabelsi et al., 2022; Veit et al., 2001) have considered agent limitations during the discovery process. (Capezzuto et al., 2021) specifies a compact formulation for multi-agent task allocation with spatial and temporal constraints. (Trabelsi et al., 2022) considers agent constraints in the form of incompatibility with some resources. The

Biswas, D.
Constraints Enabled Autonomous Agent Marketplace: Discovery and Matchmaking.
DOI: 10.5220/0012461700003636

In *Proceedings of the 16th International Conference on Agents and Artificial Intelligence (ICAART 2024) - Volume 1*, pages 396-403
ISBN: 978-989-758-680-4; ISSN: 2184-433X

authors then propose an optimal matchmaking algorithm that allows the agents to relax their restrictions, within a budget. While similar to the notion of *'bounded inconsistency'* in our specification, we bake this relaxation in the matchmaking process; rather than requiring the agents to individually relax their constraints. (Veit et al., 2001) proposes a configurable XML based framework called GRAPPA (Generic Request Architecture for Passive Provider Agents) for agent matchmaking. However, none of the above approaches consider composability of the component agent services' constraints.

In this paper, we take a holistic look at agent constraints, and their role in the discovery and matchmaking process - enabling an Autonomous AI Agent Marketplace. We outline a Predicate Logic based Constraints Model to capture/specify the constraints of services provided by an AI Agent in Section 2.

A significant contribution of this paper is the aspect of constraints composition and its impact on agent discovery. Determining the description of a complex composite service, by itself, is non-trivial (Casati et al., 2000). Given their inherent non-determinism (allowed by the "choice" operators within a composition schema), it is impossible to statically determine the actual subset of component services that would be invoked at run-time. The above implies the difficulty in selecting the component services whose constraints should be considered while defining the constraints of the composite service. Basically, the constraints of a composite service should be consistent with the constraints of its component services. In this paper, we take a bottom-up approach and discuss how the constraints of a composite service can be consistently derived from the constraints of its component services - Section 3.

We discuss how matchmaking can be performed based on the constraints model in Section 4. Current matchmaking algorithms focus on "exact" matches. They do not consider the scenario where a match does not exist. We propose to overcome the above by allowing inconsistencies during the matchmaking process (does not have to be an exact match) up to a bounded limit. Section 5 concludes the paper and provides some directions for future work.

2 CONSTRAINTS MODEL

Constraints refer to the characteristics of an agent service that need to be considered for successful execution of that service. Before proceeding, we would like to discuss some heuristics to decide if a characteristic should (or should not) be considered as a constraint. If we consider constraints as limitations, then the fact that an Airline *ABC* cannot provide booking for a particular date is also a limitation, and hence a constraint. However, we do not expect such characteristics to be expressed as constraints as they keep changing frequently. Given this, what should (or should not) be expressed as constraints is very much context-specific, and we consider constraints as a level of filtering during the discovery and matchmaking process.

An agent P provides a set of services $\{S_1, S_2, \cdots, S_n\}$. Each service S in turn has a set of associated constraints $\{C_1, C_2, \cdots, C_m\}$. The constraints are specified as logic predicates in the service description of the corresponding service published by its agent. For each constraint C of a service S, the constraint values maybe

- a single value (e.g., price of a service),
- list of values (e.g., list of destinations served by an airline), or
- range of values (e.g., minimum, maximum).

The constraint values are specified as facts containing the applicable values. More precisely, a service S provided by P having constraints $\{C_1, C_2, \cdots, C_m\}$ is specified as follows:

$$
\begin{aligned}
&S(Y, X_1, X_2, \cdots, X_m) :- \\
&\quad C_1(Y, X_1), \\
&\quad C_2(Y, list_2), member(X_2, list_2), \\
&\quad \cdots, \\
&\quad C_m(Y, minVal_m, maxVal_m), X_m \geq minVal_m, \\
&\quad X_m \leq maxVal_m. \\
&C_1(P, value(C_1)). \\
&C_2(P, list(C_2)). \\
&\cdots, \\
&C_m(P, min(C_m), max(C_m)).
\end{aligned}
$$

For example, the fact that an airline *ABC* provides vegetarian meals and has facilities for handicapped people on only some of its flights (to selected destinations) can be represented as follows:

$$
\begin{aligned}
&flight(Airlines,\ X,\ Y) :- \\
&\quad veg\_meals(Airlines,\ Destination\_List), \\
&\quad member(X,\ Destination\_List), \\
&\quad hnd\_facilities(Airlines,\ Destination\_List), \\
&\quad member(Y,\ Destination\_List). \\
&veg\_meals(ABC,\ [Paris,\ Rennes]). \\
&hnd\_facilities(ABC,\ [Paris,\ Grenoble]).
\end{aligned}
$$

We present the specification in a logic program syntax so that it can be given as input directly to a logic execution engine responsible for performing the matchmaking. Note that the logic program representation above is not the most efficient as $C_2(Y, list_2)$,

$member(X_2, list_2)$, for instance, can be easily replaced by the direct check $member(X_2, list(C_2))$. However, we intentionally keep the data (constraint values) separate (as facts) as it is important not only from a design perspective, but also required for the composition process later (Section 3).

Now, let us consider "related" constraints or scenarios where there exists a relationship among the constraints. By default, the above specification assumed an *AND* relation among the constraints (all the constraints $C_1, C_2, \cdots, C_m$ have to be satisfied). Some relationships studied in literature for the composition of logic programs are: *AND*, *OR*, *ONE-OR-MORE*, *ZERO-OR-MORE* and any nesting of the above. We only consider the relationships *AND*, *OR* and any level of nesting of both to keep the framework simple (*ONE-OR-MORE* and *ZERO-OR-MORE* can be expressed in terms of *OR*). We denote *AND* and *OR* relationships between a pair of constraints C_1 and C_2 using the logical operators $C_1 \wedge C_2$ and $C_1 \vee C_2$, respectively. The *OR* relationship between constraints $C_1 \vee C_2 \cdots \vee C_m$ of a service S provided by agent P can be specified as a logic program as follows:

$$
\begin{aligned}
&S(Y, X_1) :- \\
&\quad C_1(Y, list_1), member(X_1, list_1). \\
&S(Y, X_2) :- \\
&\quad C_2(Y, list_2), member(X_2, list_2). \\
&\cdots \\
&S(Y, X_m) :- \\
&\quad C_m(Y, list_m), member(X_m, list_m).
\end{aligned}
$$

For example: Airline *ABC* allows airport lounge access at intermediate stopovers only if the passenger holds a business class ticket or is a member of their frequent flier programme. The above scenario can be represented as follows:

$$
\begin{aligned}
&lounge\_access(Airlines,\ X) :- \\
&\quad ticket\_type(ABC,\ X,\ Business). \\
&lounge\_access(Airlines,\ Y) :- \\
&\quad frequent\_flier(Airlines,\ FF\_List), \\
&\quad member(Y,\ FF\_List).
\end{aligned}
$$

Nested *AND* and *OR* relationships among the constraints can be represented by first converting the nested relationship into a Disjunctive Normal Form (DNF), and then representing them as logic programs (based on the proposed representation mechanisms for *AND* and *OR*). More precisely, a nested *AND*/*OR* relationship among the constraints of a service S provided by agent P in DNF: $(C_1 \wedge \cdots \wedge C_k) \vee \cdots \vee (C_1 \wedge \cdots \wedge C_l)$ can be represented as a logic program as follows:

$$
\begin{aligned}
&S(Y, X_1, \cdots, X_k) :- \\
&\quad C_1(Y, list_1), member(X_1, list_1), \\
&\quad \cdots, \\
&\quad C_k(Y, list_k), member(X_k, list_k). \\
&\cdots \\
&S(Y, X_1, \cdots, X_l) :- \\
&\quad C_1(Y, list_1), member(X_1, list_1), \\
&\quad \cdots, \\
&\quad C_l(Y, list_l), member(X_l, list_l).
\end{aligned}
$$

In the next section, we discuss mechanisms to compose the constraints of a group of services, i.e., determine the constraints of a composite service, in an automated fashion.

3 CONSTRAINTS COMPOSITION

We broadly outline two composition mechanisms: Broker (Section 3.1) and Mediator (Section 3.2).

3.1 Broker

In the broker approach, the composite agent aggregates similar services offered by different providers, and provides a unique interface to them (mostly, without any modification to their functionality). In other words, the composite agent acts as a broker for the aggregated set of services.

Given this approach, let a broker B aggregate a common service S provided by agents $P_1, P_2, \cdots, P_r$. Then, the constraints specification of S offered by B can be obtained by a simple concatenation of the constraint logic program representations of S provided by $P_1, P_2, \cdots, P_r$. For example, let us consider the following scenario: A broker XYZ composing flight services offered by Airlines *ABC* and *DEF*.

$$
\begin{aligned}
&\text{Airlines } ABC \\
&flight(Airlines,\ X) :- \\
&\quad hFacilities(Airlines,\ hdList), \\
&\quad member(X,\ hdList). \\
&hFacilities(ABC,\ [Marseilles,\ Grenoble]). \\
&\text{Airlines } DEF \\
&flight(Airlines,\ X) :- \\
&\quad hFacilities(Airlines,\ hdList), \\
&\quad member(X,\ hdList). \\
&hFacilities(DEF,\ [Rennes,\ Paris]).
\end{aligned}
$$

With the above concatenation, we still have the problem that the binding returned by the matchmaking process would be *ABC*/*DEF*, and not the broker agent XYZ. A simple rewriting of *ABC* and *DEF* by XYZ does not solve the problem either, as XYZ itself would internally like to keep track of who would be

the actual agent in the event of an invocation of S. We overcome this problem by explicit assignment statements in the predicates as follows:

Composite Agent (Broker) XYZ
$flight(Airlines,\ X) :-$
$\quad hFacilities(Airlines,\ hdList),$
$\quad member(X,\ hdList),$
$\quad \text{Airlines} = XYZ.$
$hFacilities(ABC,\ [Marseilles,\ Grenoble]).$
$flight(Airlines,\ X) :-$
$\quad hFacilities(Airlines,\ hdList),$
$\quad member(X,\ hdList),$
$\quad \text{Airlines} = XYZ.$
$hFacilities(DEF,\ [Rennes,\ Paris]).$

More precisely, the constraints of services offered by broker B, aggregating service S provided by agents $P_1, P_2, \cdots, P_r$, can be derived from their respective constraints' specifications as follows:

1. Concatenate the logic program representations of the constraints of S provided by $P_1, P_2, \cdots, P_r$.
2. For each service predicate $S(Y, X_1, \cdots)$ in the concatenated logic program, add the assignment $Y = B$ in its body.

Note that this composition approach usually leads to a relaxation of constraints. In the example scenario, the composite agent XYZ would be able to offer flights with facilities for handicapped people to more destinations (Marseilles, Grenoble, Rennes and Paris), than can be offered by either the component airlines ABC (Marseilles, Grenoble) / DEF (Rennes, Paris).

An analogous example scenario if the constraint values were ranges (not lists): Let airlines ABC and DEF offer flight services (to the same destinations) at different price ranges: $(A_{min},\ A_{max})$ and $(D_{min},\ D_{max})$ respectively. Then, the broker XYZ can offer flights to the same destinations in the price range:
$(min(A_{min},\ D_{min}),\ max(A_{max},\ D_{max}))$.
Here also, the relaxation aspect of composition is observed.

3.2 Mediator

In this approach, two or more services offered by (the same or) different agents are composed to form a new composite service with some additional logic (if required). We first consider only deterministic composition (component services invoked in sequence or parallel), and then extend the mechanism to accommodate non-determinism (possibility of choice among the component services).

Given this approach, let an agent M compose composite service SC from component services $S_1, S_2, \cdots, S_n$ (provided by providers $P_1, P_2, \cdots, P_n$, respectively). For simplicity, we assume that each service is provided by a different agent. As before, we start with a concatenation of the constraint logic program representations of $S_1, S_2, \cdots, S_n$. Here, in addition, we need to provide a unified interface grouping the component services $S_1, S_2, \cdots, S_n$. For example, let us consider the following scenario: An Airline ABC with facilities for handicapped people to selected destinations,

Airline ABC
$flight(Airlines,\ X) :-$
$\quad hFacilities(Airlines,\ hdList),$
$\quad member(X,\ hdList).$
$hFacilities(ABC,\ [Marseilles,\ Grenoble]).$

and a transportation company DEF with facilities for handicapped people on its local bus networks in selected cities.

Transport DEF
$bus(Transport,\ X) :-$
$\quad hFacilities(Transport,\ ctList),$
$\quad member(X,\ ctList).$
$hFacilities(DEF,\ [Marseilles,\ Rennes]).$

Given this, the constraints of composite service provided by Travel Agent XYZ can be specified as follows:

Travel Agent XYZ
$flight\_bus(Agent,\ X) :-$
$\quad \_flight(Agent1,\ X), \_bus(Agent2,\ X),$
$\quad Agent = XYZ.$
$\_flight(Airlines,\ X) :-$
$\quad hFacilities(Airlines,\ hdList),$
$\quad member(X,\ hdList).$
$hFacilities(ABC,\ [Marseilles,\ Grenoble]).$
$\_bus(Transport,\ X) :-$
$\quad hFacilities(Transport,\ ctList),$
$\quad member(X,\ ctList).$
$hFacilities(DEF,\ [Marseilles,\ Rennes]).$

Note the $flight\_bus$ predicate representing the constraints of the newly formed composite service. Also, the primitive service predicates are prefixed with to indicate that those services are no longer available (exposed) for direct invocation, and as such their constraints are not relevant independently anymore. More precisely, the constraints of service SC composed by M can be derived from the constraints' specifications of its component services $S_1, S_2, \cdots, S_n$ as follows:

1. Concatenate the logic program representations of the constraints of $S_1, S_2, \cdots, S_n$.
2. Append the prefix to all service predicates $S_{i=1\cdots n}(Y, X_1, \cdots)$ in the concatenated logic program.
3. Add the predicate:
   $S_C(Y, X_1, \cdots, X_m) :-$
   $\_S_1(Y, X_1, \cdots, X_k), \cdots, \_S_l(Y, X_1, \cdots, X_l),$
   $Y = M.$
   where $\{X_1, \cdots, X_k\} \cup \cdots \cup \{X_1, \cdots, X_l\} = \{X_1, \cdots, X_m\}$. We refer to this predicate as the mediator predicate.

In contrast to the broker approach, this approach highlights the restrictive nature of constraints composition, especially, if the component services share a common constraint. For example, the newly composed service $flight\_bus$ can provide both flight and bus booking with facilities for handicapped people to fewer destinations (Marseilles), as compared to the destinations covered by the component services independently: $flight$ (Marseilles, Grenoble) and *bus* (Marseilles, Rennes).

An analogous example scenario if the constraint values were single values (and not lists): Let Airlines *ABC* and transport company *DEF* offer services at $\$X$ and $\$Y$ respectively, then the composite provider *XYZ* would charge $\$(X+Y)$ which is in sync with the restrictive nature of composition (the composite service costs more than any of the component services).

3.3 Non-Determinism

The above mechanism for constraints composition is sufficient if we know the set of component services that will be invoked (deterministic composition). However, the composition approach also allows a non-deterministic composition of services, where it is possible to specify a choice among the component services in the composition schema (the choice is resolved at run-time based on the current state of execution, input values, etc.).

With non-determinism, the constraints composition mechanism is much more complicated. Some of the component services, composed via choice operators, may never be invoked during an execution instance of the composite service. As such, we need some logic to determine if the constraints of a component service should (or should not) be considered while specifying the constraints of the composite service. For example, let us consider the e-shopping scenario illustrated in Fig. 1. There are two non-deterministic operators in the composition schema: Check Credit and Delivery Mode. The choice "Delivery Mode" indicates that the user can either pick-up the order directly from the store or have it shipped to his address. Given this, shipping is a non-deterministic choice and may not be invoked during the actual execution. As such, the question arises "if the constraints of the shipping agent, i.e., the fact that it can only ship to certain countries, be projected as constraints of the composite e-shopping service (or not)?". Note that even component services composed via deterministic operators (Payment and Shipping) are not guaranteed to be invoked if they are preceded by a choice. We consider approaches to accommodate this inherent non-determinism in the sequel.

Paths Based Approach: Determine all the possible execution paths based on the composition schema, and specify the constraints of the composite service in terms of the paths. The underlying intuition is that the component services in a path represent a group of component services, all or none (deterministic) of which would be invoked in an execution of the composite service. The execution paths of a composition schema given as a Directed Acyclic Graph (DAG) can be determined based on a Depth First Search (DFS) traversal of the DAG. Given this, the non-determinism is reduced to the existence of more than one possible execution path, all of which need to be represented in the constraints specification of the composite service.

More precisely, let $S_1, S_2, \cdots, S_n$ be the component services composed by agent M to provide composite service *SC*. By abuse of notation, we denote a path by the set of services in the path. Then, if
$\{S_1, \cdots, S_i\}, \cdots, \{S_1, \cdots, S_j\}$
are the possible execution paths,
$\{S_1, \cdots, S_i\} \cup \cdots \cup \{S_1, \cdots, S_j\} = \{S_1, \cdots, S_n\}$.

Given this, the constraints specification of *SC* can be derived as follows:

1. The first two steps are similar to the deterministic scenario, i.e., concatenation of the component services' constraint specifications, followed by appending of to all the service predicates $S_{i=1\cdots n}(Y, X_1, \cdots)$ in the concatenated logic program.
2. For each possible execution path $\{S_1, \cdots, S_j\}$, add the predicate:
   $S_C(Y, X_1, \cdots, X_m) :-$
   $\_S_1(Y, X_1, \cdots, X_k), \cdots, \_S_j(Y, X_1, \cdots, X_l),$
   $Y = M.$
   where $\{X_1, \cdots, X_k\} \cup \cdots \cup \{X_1, \cdots, X_l\} = \{X_1, \cdots, X_m\}$. Note that in contrast to the deterministic scenario, here we need to add a mediator predicate for each possible path.

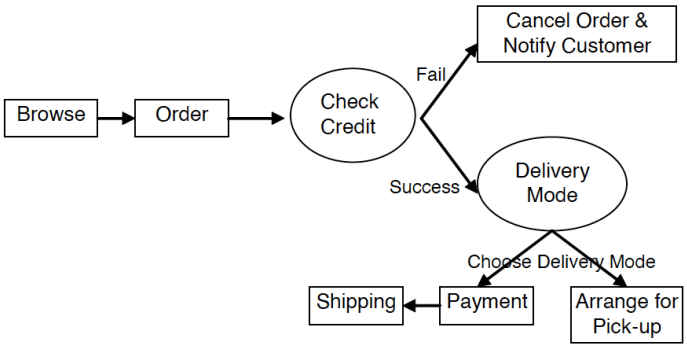

Figure 1: An e-shopping scenario.

Heuristical and Incremental Approaches. We can also decide on the set of component services whose constraints to consider in the composite service's constraints, based on some heuristics, or decide it in an incremental fashion (at run-time) as follows:

- *Optimistic*: Consider the constraints of only those component services that are guaranteed to be invoked in any execution of the composite service. The set of such services (hereafter, referred to as the *strong* set) can be determined by computing all the possible execution paths and selecting services that occur in all the paths. For example, with reference to the e-shopping scenario in Fig. 1, the strong set = {*Browse, Order*}. We call this approach optimistic as it assumes that the constraints of the component services in the strong set are sufficient to represent the constraints of the composite service.

  The concept of a strong set is analogous to the notion of strong unstable predicates (Garg and Waldecker, 1996) or predicates that will "definitely" hold (Cooper and Marzullo, 1991) in literature. For example, strong unstable predicates can be used to check if there was a point in the execution of a commit protocol when all the processes were ready to commit. Intuitively, strong unstable predicates allow us to verify that a desirable state will always occur.

- *Pessimistic*: In this approach, we take the pessimistic view and consider the constraints of all those component services that are in at least one of the possible execution paths. We refer to such a set of component services as the *weak* set. Note that the weak set would consist of all the component services if there are no "unreachable" services in the composition schema. Again, with reference to the e-shopping scenario in Fig. 1, the weak set = {*Browse, Order, Cancel Order & Notify Customer, Arrange for Pick-up, Payment, Shipping*}. We refer to this approach as pessimistic as it considers the constraints of those component services as well, that may not even be invoked during the actual execution of the composite service.

  The corresponding notion in literature is weak unstable predicates (Garg and Waldecker, 1994) or predicates that will "possibly" occur (Cooper and Marzullo, 1991). For example, weak unstable predicates can be used to verify if a distributed mutual exclusion algorithm allows more than one process to be in the critical region simultaneously. Intuitively, weak unstable predicates can be used to check if an undesirable state will ever occur.

- *Probabilistic*: Another option would be to consider the constraints of the most frequently invoked component services (or the component services in the most frequently used execution path) as the representative set of the composite service. Such a set can be determined statically from the execution logs or dynamically with the help of some mathematical model (such as, Markov Decision Processes to assign probabilities to the component services based on previous executions.

  Again, with reference to the e-shopping scenario in Fig. 1, a probable set of most frequently used component services might be {*Browse, Order, Arrange for Pick-up*}. While this option appears the most attractive at first sight, developing and solving a Markovian model (Doshi et al., 2004) is non-trivial for a complex composition schema (especially, if it involves a lot of choices).

- *Incremental*: Here, we propose an incremental approach to determining the set of component services, whose constraints to consider, while constructing the constraints of the composite service. Basically, rather than statically determining the constraints of the composite service,

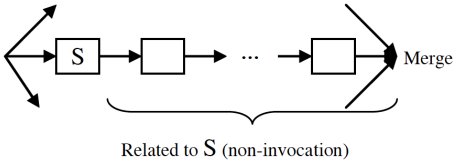

Figure 2: Related services (based on non-invocation).

this approach advocates starting with (performing matchmaking based on) the strong set, and keep on adding the constraints of "related" component services as execution progresses. We define related services as follows: For a pair of component services $S_1 \neq S_2$ of composite service S_C, S_1 and S_2 are related if and only if an invocation of S_1 implies that S_2 will be invoked as well in the same execution instance of S_C. Intuitively, if a component service S is invoked, then all the component services till the next choice in the composition schema will be definitely invoked. For example, with reference to the e-shopping scenario in Fig. 1, services *Payment* and *Shipping* are related. As mentioned earlier, the execution of both *Payment* and *Shipping* are not guaranteed. However, if *Payment* is invoked, then *Shipping* is also guaranteed to be invoked. The above definition of related services can also be extended to non-invocation of component services as follows:

Related services (extended): For a pair of component services $S_1 \neq S_2$ of composite service S_C, S_1 and S_2 are related if and only if an invocation (non-invocation) of S_1 implies that S_2 will (not) be invoked as well in the same execution instance of S_C.

Intuitively, if a component service S is not invoked, then all the component services till the next merge in the composition schema will not be invoked as well (Fig. 2). This extension is useful if we consider matchmaking for more than one composite service simultaneously (not considered here). Given this, prior knowledge that a component service will not be invoked during a particular execution instance allows better scheduling of the providers among instances.

4 AGENT MATCHMAKING

4.1 Exact Match

For a user task G, matchmaking consists of finding agents capable of executing G's (sub-)tasks. The subtasks of G might have their own constraints. Given this, the required matchmaking for G can be achieved with the help of a logic program execution engine by posing (tasks of) G's constraints as a goal against the logic program corresponding to the service constraints of the respective agents. A logic program execution engine specifies, not only if a goal can be satisfied, but also all the possible bindings for any unbounded variables of the goal. In case of multiple possible bindings (multiple agents capable of executing the same task), the agents can be ranked using some user defined preference criteria to select the most optimum among them.

4.2 Approximate Match

In this section, we consider the scenario where matchmaking was unsuccessful, i.e., there does not exist a set of agents capable of executing the given task G. Given this, it makes sense to allow some inconsistency while selecting an agent. Note that inconsistency is often allowed by real-life systems, e.g., flight reservation systems allow flights to be overbooked, but only up to a limited number of seats. Thus, the key here is *bounded inconsistency*. Basically, for a goal $G = \{T_1, T_2 \cdots, T_n\}$, the selected agent for one of the tasks T_i does not have to be a perfect match as long as their accumulated inconsistency is within a specified limit. Note that the inconsistency induced by an agent may also have a counter effect on (reduce) the inconsistency induced by another task T_j. For a given goal $G = \{T_1, T_2 \cdots, T_n\}$, approximate matchmaking can be achieved as follows:

1. Determine the common constraints: A constraint C is common with respect to G, if more than one task of G has constraints based on C. For example, if tasks T_1 and T_2 need to be completed within 3 and 4 days respectively, then they have a common time based constraint. Studies have shown that most constraints in real-life scenarios are based on the constraints: location, price, quantity or time.
2. For each common constraint C, define a temporary variable q_C (to keep track of the inconsistency with respect to C).
   Initially, $q_C = 0$.
3. For each task T_i and a common constraint C: Let vC_i denote the constraint value of T_i with respect to C. For example, $vTime_1 = 3$ denotes the completion time constraint value of T_1.
   Delete the C constraint of T_i from the constraints specification of G.
4. Perform matchmaking based on the reduced goal (with the common constraint predicates deleted in the above step).

5. If the matchmaking was successful: [Note that if matchmaking was unsuccessful for the reduced goal, then it would definitely have been unsuccessful for the original goal.] Let $P(T_i)$ denote the agent selected to execute T_i.

   For each deleted common constraint C of T_i (Step 3), get the best possible constraint value $vBestC_i$ of $P(T_i)$, and compute $q_C = q_C + (vC_i - vBestC_1)$.

   For example, let us assume that $P(T_1)$ and $P(T_2)$ need at least 5 and 1 days, respectively to complete their work. Given this, $q_t = 0 + (vC_1 - vBestC_1) + (vC_2 - vBestC_2) = (3-5) + (4-1) = 1$.

6. The matchmaking results are valid if and only if for each common constraint $C, q_C > 0$. For example, $P(T_1)$ and $P(T_2)$ are valid matches for the tasks T_1 and T_2 respectively, as $q_t > 0$.

Note that this matchmaking would not have been possible without the (approximate) extension as $P(T_1)$ violates (takes 5 days) the completion time constraint (3 days) of T_1. For simplicity, we have only considered numeric value based constraints in the above algorithm.

5 CONCLUSION

In this paper, we focused on the discovery aspect of Autonomous AI Agents. To execute a complex task, a pre-requisite is a marketplace with a registry of agents, specifying their service(s) capabilities and constraints. We outlined a constraints based model to specify agent services. To enable hierarchical composition, we showed how the constraints of a composite agent service can be derived and described in a manner consistent with respect to the constraints of its component services. We proposed a paths based approach, as well as heuristical (optimistic, pessimistic, probabilistic) and incremental (relative) approaches, to accommodate the inherent non-determinism. Finally, we discussed approximate matchmaking, and showed how the notion of bounded inconsistency can be leveraged to discover agents more efficiently.

In future, it would be interesting to extend the matchmaking algorithm to simultaneous discovery of more than one user request. Doing so, leads to some interesting issues like efficient scheduling of the available agents (touched upon briefly in Section 3.2). We would also like to consider the top-down aspect of constraints composition, i.e., to define the constraints of a composite service independently and verifying their consistency against the constraints of its corresponding component services.

REFERENCES

(2000). *Universal Description, Discovery, and Integration (UDDI) Technical White Paper*. UDDI.org. http://www.uddi.org/pubs/Iru_UDDI_Technical_White_Paper.pdf.

(2023). *AutoGPT: the heart of the open-source agent ecosystem*. AutoGPT. https://github.com/Significant-Gravitas/Auto-GPT.

(2023). *Five Best Practices for Building an Effective API Marketplace*. MuleSoft. https://www.mulesoft.com/api-university/five-best-practices-building-effective-api-marketplace.

(2023). *NexusGPT: World's 1st AI-freelancer platform*. NexusGPT. https://nexus.snikpic.io/.

Bordes, A., Boureau, Y.-L., and Weston, J. (2017). Learning end-to-end goal-oriented dialog.

Capezzuto, L., Tarapore, D., and Ramchurn, S. D. (2021). Large-scale, dynamic and distributed coalition formation with spatial and temporal constraints. In *Multi-Agent Systems*, pages 108–125. Springer International Publishing.

Casati, F., Ilnicki, S., Jin, L., Krishnamoorthy, V., and Shan, M.-C. (2000). Adaptive and dynamic service composition in eflow. In *Advanced Information Systems Engineering*, pages 13–31, Berlin, Heidelberg. Springer Berlin Heidelberg.

Cooper, R. and Marzullo, K. (1991). Consistent detection of global predicates. *SIGPLAN Not.*, 26(12):167–174.

Doshi, P., Goodwin, R., Akkiraju, R., and Verma, K. (2004). Dynamic workflow composition using markov decision processes. In *Proceedings. IEEE International Conference on Web Services, 2004.*, pages 576–582.

Garg, V. and Waldecker, B. (1994). Detection of weak unstable predicates in distributed programs. *IEEE Transactions on Parallel and Distributed Systems*, 5(3):299–307.

Garg, V. and Waldecker, B. (1996). Detection of strong unstable predicates in distributed programs. *IEEE Transactions on Parallel and Distributed Systems*, 7(12):1323–1333.

Park, J. S., O'Brien, J. C., Cai, C. J., Morris, M. R., Liang, P., and Bernstein, M. S. (2023). Generative agents: Interactive simulacra of human behavior.

Trabelsi, Y., Adiga, A., Kraus, S., and Ravi, S. S. (2022). Resource allocation to agents with restrictions: Maximizing likelihood with minimum compromise. In *Multi-Agent Systems*, pages 403–420. Springer International Publishing.

Veit, D., Müller, J. P., Schneider, M., and Fiehn, B. (2001). Matchmaking for autonomous agents in electronic marketplaces. In *Proceedings of the Fifth International Conference on Autonomous Agents*, page 65–66. Association for Computing Machinery.

Weiss, G. (2016). *Multiagent Systems, Second Edition*. Intelligent Robotics and Autonomous Agents. MIT Press, 2nd edition.

Yan, J., Hu, D., Liao, S. S., and Wang, H. (2015). Mining agents' goals in agent-oriented business processes. *ACM Trans. Manage. Inf. Syst.*, 5(4).