Artificial Intelligence 236 (2016) 30–64

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint

Domain-independent planning for services in uncertain and dynamic environments

Eirini Kaldeli [a,b], Alexander Lazovik [b], Marco Aiello [b]

[a] School of Electrical and Computer Engineering, National Technical University of Athens, Zographou Campus, 15780 Athens, Greece
[b] Johann Bernoulli Institute, University of Groningen, Nijenborgh 9, 9747 AG Groningen, The Netherlands

ARTICLE INFO

Article history:
Received 12 May 2014
Received in revised form 6 March 2016
Accepted 9 March 2016
Available online 14 March 2016

Keywords:
AI planning
Web service composition

ABSTRACT

Research in automated planning provides novel insights into service composition and contributes towards the provision of automatic compositions which adapt to changing user needs and environmental conditions. Most of the existing planning approaches to aggregating services, however, suffer from one or more of the following limitations: they are not domain-independent, cannot efficiently deal with numeric-valued variables, especially sensing outcomes or operator inputs, and they disregard recovery from runtime contingencies due to erroneous service behavior or exogenous events that interfere with plan execution. We present the RuGPlanner, which models the planning task as a Constraint Satisfaction Problem. In order to address the requirements put forward by service domains, the RuGPlanner is endowed with a number of special features. These include a knowledge-level representation to model uncertainty about the initial state and the outcome of sensing actions, and efficient handling of numeric-valued variables, inputs to actions or observational effects. In addition, it generates plans with a high level of parallelism, it supports a rich declarative language for expressing extended goals, and allows for continual plan revision to deal with sensing outputs, failures, long response times or timeouts, as well as the activities of external agents. The proposed planning framework is evaluated based on a number of scenarios to demonstrate its feasibility and efficiency in several planning domains and execution circumstances which reflect concerns from different service environments.

1. Introduction

Software service infrastructures enable the large scale integration of heterogeneous systems and solve a number of interoperability issues. A prototypical example is that of *Web Services* (WS) where programmatic access to Web resources is guaranteed via standardized XML interfaces, such as those defined by the Web Service Description Language (WSDL) (www.w3.org/TR/wsdl). Automated planning can contribute to the realization of service infrastructures that go beyond basic interoperation and ad hoc process specifications, offering highly automated functionalities that are adaptable to changing user needs and environmental conditions. The goal is to compose and interact automatically with several service providers in order to offer value added functionalities. More precisely, *a service composition* is a combination of operations provided by different services to satisfy complex objectives which cannot be fulfilled by a single service instance. *Planning* is the process of "choosing and organizing *actions* by anticipating their expected outcomes," with the aim of achieving a pre-stated

E-mail address: kaldeli@gmail.com (E. Kaldeli).

http://dx.doi.org/10.1016/j.artint.2016.03.002
0004-3702/

goal [42]. The analogies between the problem of Web Service composition (WSC) and automated planning are evident and have been exploited before (e.g., [1,108,98,110,15]): actions correspond to functionalities offered by different services, and the goal is derived from a user request or inferred by a situation that calls for a combination of services.

The composition method advocated herein is driven by the general aim of combining services automatically and on-demand, relying solely on individual descriptions of loosely-coupled software components, and a declarative goal language. The idea is to maintain a generic and modular repository that comprises a number of atomic service operations, from booking flights to arranging appointments with a doctor, and that can serve a variety of objectives with minimal request-specific configuration. We use *domain-independent planning* and propose an extended language for expressing complex goals in a declarative fashion, detached from the particularities and interdependencies of the available services. This is unlike most previous approaches that restrict the applicability of the domain to a set of anticipated user needs, predefined in the form of some procedural template, e.g., [90,108].

1.1. Characteristics of service domains

Service domains are *data intensive*, i.e., they deal with variables ranging over very large domains, such as prices, dates, product quantities, and so on. If the behavior of a service operation is to be modeled by a planning operator, then this must quite frequently involve fluents with numeric-valued input arguments. For example, an operation to reserve an airline ticket is parametrized by data associated with the flight. Whether or not the application of such an operation has the intended effect depends on the choice of the correct value of the input parameters. One must also take into account numeric properties (e.g., the temperature must be greater than 0) both in preconditions and in the goal, and be able to apply arithmetic operations (e.g., increase an account balance by a certain amount). Things become more complicated when one considers that numeric information is very often the output of *sensing* operations. Due to the many operators involving variables taking values in domains with large cardinality, either as input parameters or as outputs, the number of ground operators and, as a consequence, the size of the search space can increase enormously.

In a service environment, the planner must deal with *uncertainty* about the initial state, i.e., consider that there is a number of possibilities about the actual values of certain variables. The actual value of an unknown variable can be returned after the invocation of a sensing operator, referred to as *knowledge-gathering* or observational, which returns exactly one outcome out of a (possibly very large) set of choices. In fact, marketplaces consisting of services publicly available on the Web are usually dominated by services that are data sources [31], which in a planning context are modeled as sensing operators. A successful plan may be conditioned on the outcomes of such actions, e.g., the user may want to go ahead with buying a book from amazon.com if this costs less than a maximum price. In a domain-independent setting, the planning agent is required to *plan for sensing*; the planner should be able to identify which knowledge it lacks for satisfying the goal, and reason about how to find it, instead of relying on pre-specified queries [110,76,4]. The planner should also proactively see to the data flow of the plan, i.e., the way the service return values are used by subsequent actions. For example, a user may want to mail a parcel to someone whose address he does not already know. The plan should thus first consult a white pages service, and then give the order for posting by passing the right address value to the respective input parameter. For data intensive service domains, determining the parameters for an action can be just as difficult as determining which actions belong to the plan. Since almost all state-of-the-art planners resort to some kind of pre-processing for compiling the PDDL [87] domain into a fully grounded encoding, on-the-fly handling of runtime outputs is difficult to implement.

Uncertainty applies, however, not only to the initial state and the non-determinism of the many possible values of the outputs of sensing actions. In fact, a service invocation may behave in unexpected ways: i mat return a failure, not respond at all, or even act in a way different from the one prescribed by its description. The actual outcomes of a service invocation only become visible at *runtime*, when the plan is exposed to the actual environmental conditions. Thus, the problem of uncertainty is directly related to the interaction between planning and execution.

The state of a service domain may not only change as a result of the deliberate actions executed by the planning agent, but also due to the activity of other exogenous agents which are active in the same environment. The activity of other actors may have repercussions for the composition-plan under execution, and render it invalid, either because information own which it relies has meanwhile become obsolete, or because certain scheduled actions are no longer applicable under the new circumstances. For example, considering a partially executed composition which involves circumnavigation of a robot, if an external actor puts an obstacle in the robot's way in the middle of execution, the robot may fall unless it revises its decisions about how to move. With a few notable exceptions [4,17,73], context *dynamicity* has largely been overlooked by existing planning approaches to WSC. Moreover, in many service environments, dynamicity also applies to the availability of information and services; e.g., the services offered by a mobile phone may appear and disappear depending on the location of the phone.

1.2. Approach

We choose to work directly at the schematic level of the planning domain, i.e., without performing any grounding. Following the so-called Multi-Valued Task (MPT) paradigm [53], we use state variables rather than predicates as the basic elements for describing the world. According to that view, a state is a tuple of values to variables, leading to a more compact encoding. In the remainder of the paper, we use the terms *planning operator* and *action* interchangeably. This should be kept

in mind when comparing with other planning approaches to service composition, which use "number of actions" to refer to the number of grounded operators (the grounded task is exponentially larger than the schematic one).

Our proposal is realized in the RuGPlanner, which is based on translating the domain and the goal into a *Constraint Satisfaction Problem* (CSP) and applying a state of the art constraint solver to compute a solution-plan. Planning as CSP fits well with many aspects that are of particular concern for service domains. The encoding expected from the constraint solver suits the MPT-like schematic planning representation, and constraint solving supports the efficient handling of numeric expressions and variables ranging over large domains. Moreover, such an encoding is well suited to most standard WS description languages, such as WSDL, which are based on state variables rather than predicates; complex goals can also be expressed in the form of constraints. Finally, plans can include parallel actions, which are particularly useful at execution time.

To deal with uncertainty and the possible mismatch between off-line planning and actual execution results, our proposal relies on *continual planning*: upcoming plan steps anticipated offline can be revised as execution proceeds in the face of inconsistencies that stem either from the newly sensed information, erroneous service behavior or the actions of exogenous agents. We describe this alternating sequence of offline planning and online execution as *orchestration*, a well-known concept in the service-oriented computing community which denotes how the execution of a composition is managed. In a service infrastructure, the orchestration engine is a central coordinator which interacts with the component WSs in accordance with the composite process specifications. We reserve a similar role for an orchestration component that interacts with the environment, informs the planner about the data it has collected, and decides when to switch from planning to execution and vice versa. Depending on the situation, *re-using* parts of the existing plan may speed up the process of plan revision. Moreover, a non-blocking strategy is adopted with respect to waiting for the response of sensing actions, so that the framework can go on with the planning and execution of actions that do not depend on the expected response. Continual planning can be nicely incorporated into CSP. In fact, dynamic constraint solving allows for the efficient addition and removal of constraints. This enables the planner to constantly incorporate new facts about the environment or remove obsolete ones, check for possible inconsistencies, and react accordingly.

1.3. A motivating example

Suppose that a user is happy to learn that in the coming days a singer she likes is making a tour in the country where she lives. She would like to book a ticket and a hotel room for the nearest upcoming concert whose date and location meet criteria including weather conditions, the distance from her hometown, her availability according to her on-line agenda, and the price she is willing to pay for an overnight stay.

These requirements are expressed in the form of a declarative extended goal. Satisfaction of this goal requires collaboration of services coming from diverse business domains—related to traveling, entertainment events, maps, calendars and weather services—in a manner that can hardly be anticipated. Depending on the information returned at runtime, there are clearly many possible ways for this goal to be fulfilled. For example, it may turn out that the location of the first upcoming concert is too far, or that on that date there is no hotel available within her budget. In such cases, the original plan must be interrupted and revised; the conditions regarding the whereabouts and date of the next concert will need to be looked up. To further complicate things, at any moment a service may fail. So, if, e.g., the booking service of the first hotel that meets the user's criteria happens to be in a permanent state of failure, an alternative hotel will need to be looked for, and, depending on the result, the goal may finally be satisfied or not.

1.4. Content and organization

This paper provides a unified and comprehensive approach to domain-independent planning via CSP, with applications to service composition in uncertain and dynamic environments. The manuscript extends preliminary ideas about service composition via CSP-based planning with extended goals which were first presented in [66]. An orchestration algorithm based on replanning from scratch has been described in [67]. The offline planning framework has been used and evaluated in two diverse settings: an infrastructure for smart homes with intelligent devices exposed as services [69,119,24,68], and a platform for Business Process recovery in case of processes interference [114]. However, these papers do not present the methodological foundations of the planning system, which is treated as a "black box" being part of a service-oriented architecture, but rather describe specific applications. In the present paper, we set the foundation for the applications presented in our previous papers and we describe, for the first time, the formal methods (algorithms, syntax, semantics) behind the RuG planning system. We also introduce a novel orchestration algorithm based on plan revision by continually altering the CSP. Finally, a new and extensive set of evaluation experiments that provides an insight into the behavior and performance of the RuGPlanner is presented here.

The remainder of this paper is organized as follows. Section 2 contains the basic definitions and algorithms that pertain to the offline working of the RuGPlanner system. We describe the representation of the planning domain extended with additional variables to model the knowledge-level representation, its transformation into a CSP, as well as the syntax and the semantics of the language for expressing extended goals. How the resulting CSP is solved by a constraint solver and an example of an optimistic plan produced by the offline planner are presented in Section 3. The interesting case where offline planning has to be interleaved with execution is investigated in Section 4. The focus is the design and implementation of an

orchestration framework, which is characterized by a high level of non-blocking concurrency and can deal with a number of inconsistencies that arise due to the uncertain and dynamic nature of service environment: with sensing outputs that violate the optimistic offline assumptions, with erroneous service behaviors that contradict the expected effects, with long response times, and with exogenous events that interfere with the plan execution. A discussion of the orchestration policies and of the implementation is provided in Section 5. The overall framework is evaluated on a number of scenarios in Section 6, which demonstrate the instantiation of output-to-input parameters matchings, the tradeoff between plan refinement and planning from scratch, and the case of dealing with actions that take too long to respond. To conclude, Section 7 gives an overview of the related work and concluding remarks are presented in Section 8. Appendix A details the orchestration algorithm.

2. The RuGPlanner

In general, under conditions of uncertainty about the initial state, the search space is no longer the set of states of the domain, but its power set. The planner can resort to sensing operations to acquire all the knowledge it misses and which is necessary to fulfill a goal. Sensing operations return exactly one outcome amongst a (possibly very large) set of deterministic choices. In that respect, a plan computed offline represents a traversal between sets of states rather than complete descriptions of states, and has only the *potential* to achieve the goal, if there is *some* state sequence that could arise from the plan's execution and satisfies the goal. Such a plan is usually referred to as *weak* [27]. Finding such a context-dependent plan is a much simpler task than computing a strong contingent plan with conditional branches which would satisfy the goal in *all* possible state sequences that could arise from observational effects. Postponing the computation of alternative contingent branches till more information is acquired from the environment is a more feasible approach for WS scenarios that involve many numeric variables and output-to-input mappings, and being optimistic is likely to be paid off. Thus, all sources of non-determinism, where the actual behavior of actions at execution time contradicts the expected effects as modeled in the planning domain or external agents alter the world in unanticipated ways, are left to be treated by interleaving planning with execution and performing continual planning as described in Section 4.

Although planning as CSP is not yet as competitive as modern best-performing domain-independent planners in the International Planning Competitions (IPC), it should be noted that service environments put forward rather different challenges than the domains used in the IPC. In service environments, complexity does not usually stem from the need for complex combinatorial propositional reasoning, since there are fewer interdependencies between different actions/service operations [31] – for example, contrast the broadly used travel domain [1] with the PDDL domains in IPC (ipc.icaps-conference.org). The main challenges come rather from other sources, such as uncertainty and sensing regarding numeric-valued variables, the expressive power of the domain representation, the dynamic nature of context, and the support for complex goals. In the following, we present how the CSP-based RuGPlanner meets these requirements, providing for a highly expressive action schema, endowed with a number of features, mostly overlooked up to now. These include the support for parallel actions, which are very important given the large response times of many operations (especially sensing ones), handling of numeric variables and input parameters, numeric preconditions, and a large variety of effects. A powerful language for expressing complex goals, which accommodate for ordering constraints, maintainability properties, numeric expressions and hands-off requirements, is supported. The planning problem modeling accommodates for incomplete knowledge and information-gathering, which is realized by an intuitive knowledge-level representation which is automatically generated, given the high-level description of the domain and the goal.

2.1. Planning domain

To deal with incomplete knowledge and sensing, the planning domain description is enriched with a knowledge-level representation to model observational actions (sensing effects). Conditional effects are also provided. Moreover, the planning formalism accommodates for numeric functions and effects beyond mere assignments, such as increase/decrease. Another characteristic of the planning schema is that input arguments to actions may range over numeric-valued domains just as all other variables. Since the planning problem is translated into a constraint solver, the supported variable domains (integer, real, lower and upper bounds) depend on the underlying constraint solver that is used (see Section 5.5 for more details). The planning domain accommodated by the RuGPlanner is described in Definition 1.

Definition 1 *(Planning Domain (PD)).* A Planning Domain is a triple $\mathcal{PD} = \langle Var, Par, Act \rangle$, where:

- Var is a finite non-empty set of variables. Each variable $v \in Var$ ranges over a finite domain D^v.
- Par is a finite non-empty set of variables that play the role of input parameters to members of Act. Each variable $p \in Par$ ranges over a finite domain D^p. Par and Var are disjoint sets.
- Act is the set of actions. An action $a \in Act$ is a quadruple $a = \langle id(a),\ in(a), precond(a), effects(a) \rangle$, where:
  - $id(a)$ is a unique identifier
  - $in(a) \subseteq Par$ are the input parameters of a
  - $precond(a)$ is a propositional formula over $Var \cup Par$, which conforms to the following syntax:
    $precond(a) ::= prop \mid precond(a) \wedge precond(a) \mid precond(a) \vee precond(a) \mid \neg precond(a)$

$prop ::= var \circ val \mid var_1 \circ var_2 \mid var_1 \diamond var_2) \circ val \mid known(var) \mid brel(var_1, \ldots, var_n)$
where $var, var_1, \ldots, var_n \in (Var \cup in(a))$, val is some constant, $\circ$ is a relational operator ($\circ \in \{=, <, >, \neq, \leq, \geq\}$), $\diamond$ a binary operator ($\diamond \in \{+, -\}$), $known(var)$ a boolean relation indicating that var is known, and $brel$ an n-ary Boolean relation. We write $\bigwedge_i precond_i(a)$ to denote a sequence of conjunctions on preconditions, and likewise $\bigvee_i precond_i(a)$ for disjunctions.

- $effects(a)$ is a conjunction of any of the following elements:
  - $assign(var, v)$, where v is some constant or $v \in Var$
  - $assign(var, f(v_1, v_2))$, where $v_1, v_2 \in (Var \cup in(a))$ or v_1, v_2 are constants, and f the addition or the subtraction function
  - $increase(var, v)$ or $decrease(var, v)$, where $v \in Var \cup in(a)$ or v is some constant
  - $sense(var)$, where $var \in Var$
  - $cond\_effect(prop, effect(a))$, which models a conditional effect, that is applied at the next state only if $prop$ holds at the current state
  - $sense\text{-}cond\_effect(prop, effect(a))$, which models a conditional effect applied at the next state only if $prop$ holds at the next state. Used for effects that should materialize only if the outcome of a sensing effect satisfies $prop$.

Example 1 illustrates three examples of actions expressed in terms of preconditions and effects. Variables in Var can only change values as the result of an action, while input parameters are left free to be assigned any value by the planner. A planning state s is defined as a relation $s = \{(x, D_s^x) \mid \forall x \in Var \cup Par\}$, where $D_s^x \subseteq D^x$, and D^x is the domain of x. Thus, the notion of state adopted herein encompasses a *set* of traditional states representing assignments of values to the variables, and allows us to accommodate for incomplete knowledge. The domain of x at s is given by the *state-variable* function $[\![x]\!](s)$, so that $[\![x]\!](s) = D_s^x$ if $(x, D_s^x) \in s$. If $|D_s^x| = 1$, x at s has a specific value. An action a is applicable on s if its preconditions hold at s, and its execution leads to a successor state s'. The propositions in $precond(a)$ refer to the values of variables Var and parameters Par at state s, whereas the updates instructed by $effects(a)$ refer to the variables Var at state s'. We say that $precond(a)$ holds at s (or alternatively that s *satisfies* $precond(a)$) if $precond(a)$ evaluates to true for *all* possible assignments to values consistent with the domains of the variables at s. This implies for example that given a state $s = \{(v_1, \{1\}), (v_2, \{1, 2\})\}$ and an action a which has a precondition $v_1 = v_2$, a is not applicable at s. As we see in Section 2.2, $effects(a)$ also amount to a conjunction of propositions that should hold at s'.

The effects $sense(var)$ are called *observational* or knowledge-providing, i.e., they observe the current value of a variable, while the other effects are *world-altering*, i.e., they change the value of a variable. Variables that are part of sensing effects correspond to WS outputs (e.g., indicated with respective annotations in WSDL documents). An action may have both observational and world-altering effects. To provide for incomplete information and sensing, the domain is extended by additional variables to model the knowledge-level representation and distinguish between sensing and world-altering actions. These variables are generated automatically given a planning domain $\mathcal{PD}$. First, for each $var \in Var \cup Par$, a new boolean variable var_known is introduced, which indicates whether var is known at state s ($[\![var\_known]\!](s) = true$) or not ($[\![var\_known]\!](s) = false$). If $known(var)$ holds at state s, this is equivalent to $[\![var\_known]\!](s) = true$. The role of these knowledge-base variables becomes evident when looking at how effects and goals are translated into constraints. For every $kvar \in Var$ that participates in an observational effect, we introduce a new variable $kvar\_response$, which is a placeholder for the value returned by the respective sensing operation. Since this value is unknown until execution time, $kvar\_response$ ranges over $kvar$'s domain ($kvar\_response \in D^{kvar}$). We also maintain for every variable $cvar \in Var$ that is part of at least one world-altering effect a boolean flag $var\_changed$, which becomes true whenever this effect takes place. Thus, we end up with an extended set of variables $V = Var \cup Kb \cup Cv \cup Rv$, where Kb is the set of knowledge-base variables, Cv the set of the change-indicative variables, and Rv the response variables. States are also extended to include all variables in $V \cup Par$.

On top of the action descriptions in $\mathcal{PD}$, there may be a set of general constraints, which capture a simple aspect of the ramification problem [33], i.e., indirect effects of actions on variables. For the RuGPlanner, a general constraint is an implication constraint stating that if a variable $var_1 \in Var$ has some specific value(s), then a unique value of $var_2 \in Var$ can be concluded as well. A general constraint has the form $\bigvee_i var_1 = v_i \Rightarrow var_2 = v$, where v_i are some constants $v_i \in D^{var_1}$ and $v \in D^{var_2}$. For example, let us consider an action which moves an actor between certain rooms (see Example 1). The effect of the action refers to an assignment to the variable *robotLoc* which indicates the location of the moving robot that may also imply a change to variable *robotRoom*. The latter can be modeled as a function of the former, since knowing the location, we can infer the room.

2.2. Encoding the planning domain into a CSP

Following a common practice, we consider a *bounded* planning problem, i.e., we restrict our target to finding a plan of length at most k. Next, we illustrate how the service domain is encoded into a CSP, for some given integer k. The process is similar to the one described in [42] (alternative encodings based on the planning graph are proposed in [70,29]). A constraint satisfaction problem and its solution are defined as follows:

Definition 2 *(CSP).* A *Constraint Satisfaction Problem* is a triple $CSP = \langle X, \mathcal{D}, C \rangle$ where:

- $X = \{x_1, \ldots, x_n\}$ is a finite set of n variables.
- $\mathcal{D} = \{D^1, \ldots, D^n\}$ is the set of finite domains of the variables in X, so that $x_i \in D^i$.

- $\mathcal{C} = \{c_1, \ldots, c_m\}$ is a finite set of constraints over the variables in X. A constraint c_i involving a subset of variables in X is a proposition that restricts the allowable values of its variables.

Definition 3 *(Solution to a CSP).* A solution to a $CSP = \langle X, \mathcal{D}, \mathcal{C} \rangle$ is an assignment of values to the variables in X, $\{x_1 = v_1, \ldots, x_n = v_n\}$, with $v_i \in D^i$, that satisfies all constraints in $\mathcal{C}$.

Given a planning domain extended with the knowledge-level representation $\mathcal{PD}' = \langle V, Par, Act \rangle$, the aim is to encode $\mathcal{PD}'$ into a $CSP = \langle X, \mathcal{D}, \mathcal{C} \rangle$. First, the variables X are derived as follows: for each variable $x \in V \cup Par$ ranging over D^x, and for each $0 \leq i \leq k$, we define a CSP variable $x[i]$ in CSP with domain D^x. Actions are also represented as variables: for each action $a \in Act$ and for each $0 \leq i \leq k-1$, a boolean variable $a[i]$ is defined. This way the computed plan can include parallel actions, a fact that saves time at execution.

After having derived the CSP state variables X, the actions' preconditions and effects are encoded into constraints. Given an action $a = (id(a), in(a), precond(a), effects(a))$, we use the notation $precond(a)[i]$, $prop[i]$ and $effect(a)[i]$ to indicate the preconditions, propositions and effects on the state variables corresponding to state i. Thus, $precond(a)[i]$ ($effect(a)[i]$ respectively) results from substituting every variable $x \in X$ which appears in $precond(a)$ ($effect(a)$) by its corresponding state variable $x[i]$.

Constraint encoding

For each action a, and for each $0 \leq i < k$:

- We add the constraint $a[i] = true \Rightarrow precond(a)[i] \wedge \bigwedge_{v \in precond(a)} v\_known[i] = true$
- We add a constraint which enforces that all input parameters should be known: $a[i] = true \Rightarrow \bigwedge_{p \in in(a)} p\_known[i] = true$
- For every $effect_j$ in $effects(a)$, we add a constraint of type $a[i] = true \Rightarrow \bigwedge_j constr(effect_j)[i+1]$, where $constr(effect_j)[i+1]$ is a constraint derived depending on the type of $effect_j$:
- Case $assign(var, v)$:
  - $var[i+1] = v[i] \wedge var\_know[i+1] = true \wedge var\_changed[i+1] = true \wedge v\_known[i] = true$, if $v \in Var \cup in(a)$
  - $var[i+1] = v \wedge var\_know[i+1] = true \wedge var\_changed[i+1] = true$, if v is some constant

  Similarly, for the effect of type $assign(var, f(v_1, v_2))$
- Case $increase(var, v)$:
  - $var[i+1] = var[i] + v[i] \wedge var\_changed[i+1] = true \wedge v\_known[i] = true$, if $v \in Var \cup in(a)$
  - $var[i+1] = var[i] + v \wedge var\_changed[i+1] = true$, if v is some constant

  Similarly, for the effect of type $decrease(var, v)$
- Case $sense(var)$: $var[i+1] = var\_response[i+1] \wedge var\_known[i+1] = true$
- Case $cond\_effect(prop, effect)$: $prop[i] \Rightarrow constr(effect)[i+1]$
- Case $sense\text{-}cond\_effect(prop, effect)$: $prop[i+1] \Rightarrow constr(effect)[i+1]$

The translation of effects into constraints may entail the addition of constraints that should hold at the previous state (precondition). For example, to assign some variable v to some other variable var, v should already be known. In cases where $precond[i]$ includes a Boolean relation $brel(var_1, \ldots, var_n)[i]$, this is substituted by a proposition on these variables, according to translation rules specific to the relation $brel$. Depending on the relation, the resulting constraints may be less or more complex. For example, *adjacent_same_room(loc1, loc2)* (see Example 1) is translated into a disjunction which includes all possible allowed value pairs of variables $loc1, loc2$: $\bigvee_{c1_i, c2_j}(loc1 = c1_i \wedge loc2 = c2_j)$, for all location values $c1_i, c2_j$ which are adjacent and belong to the same room. Thus, in cases of boolean relations whose evaluation to true or false requires the knowledge of all the possible combinations of value assignments, and can only be expressed in the form of constraints by an exhaustive enumeration of these combinations, grounding is not avoided.

On top of the domain description, restrictions referring to the initial state are expressed in the form of a conjunction of propositions $\bigwedge_i prop\_init_i$, where $prop\_init_i$ are propositions on the variables $x \in X$. The encoding of the (partial) description of the initial state corresponds to the addition of the constraints $prop\_init[0]$ for each proposition $prop\_init$ that refers to the initial state.

A strong requirement of knowing all variables involved in the preconditions is also added as part of the precondition constraints. This ensures that the preconditions hold for all possible assignments to variables consistent with their allowed domain at a given state. On the other hand, this excludes the applicability of an action in some cases in which it would be admissible. For example, given a state $s = \{(v, \{1, 2\}), (v\_known, \{false\})\}$ and an action a with precondition $v < 3$, a cannot be applied at s. This restriction is necessary to prohibit undesirable situations, such as allowing the application of a at

$s = \{(v, \{1, 2, 3\}), (v\_known, \{false\})\}$. In this case, the constraint solver would be able to find *some* assignment that satisfies the constraints, which, however, is undesirable, since we cannot be sure if the application of a is safe, given the uncertainty about v's actual value. Only if there is a way to sense v, should the application of a be permitted. This restriction implies that the RuGPlanner is not able to handle problems with partial observability such as the ones illustrated in [99], where actions can be applied even if some variable in the preconditions is unknown.

The general constraints are also translated at the level of CSP variables as constraints that should hold at all states.

For each general constraint $\bigvee_i (var_1 = v_j) \Rightarrow var_2 = v$ and for each $0 \leq i < k$ the following constraint is added: $\bigvee_i var_1[i] = v_j \Rightarrow var_2[i] = v \ \wedge \ var_2\_known[i] = true$

Example 1 shows the constraints associated with three actions, as a result of the constraint encoding process.

Example 1

findAccBalance(accIdPar1)
`prec:` $\emptyset$
`effects:` $sense(accBalance)$

CSP constraints $\forall_{0 \leq i < k}$:
`prec constraints:` /*parameters known*/
$findAccBalance[i] = true \Rightarrow accIdPar1\_known[i] = true$
`effect constraints:` /*sensing*/
$findAccBalance[i] = true \Rightarrow accBalance\_known[i+1] = true \wedge$
$accBalance[i+1] = accBalance\_response[i+1]$

payIn(amountPar2, accIdPar2)
`prec:` $\emptyset$
`effects:` $increase(accBalance, amountPar2)$

CSP constraints, $\forall_{0 \leq i < k}$:
`prec constraints:` /*parameters known*/
$payIn[i] = true \Rightarrow \ (amountPar2\_known[i] = true \wedge accIdPar2\_known[i] = true)$
`effect constraints:` /*world-altering*/
$payIn[i] = true \Rightarrow accBalance\_changed[i+1] = true \wedge \ accBalance[i+1] = accBalance[i] + amountPar2[i]$

moveRobot(robotLocPar, robotRoomPar)
`prec:` $robotLoc \neq robotLocPar \wedge \ (adjacent\_same\_room(robotLoc, robotLocPar) \vee$
$(adjacent\_diff\_rooms(robotLoc, robotLocPar) \wedge \ door\_open(robotRoom, robotRoomPar)))$
`effects:` $assign(robotLoc, robotLocPar)$

Location – room general constraints
$robotLoc = LOC00 \vee robotLoc = LOC01 \Rightarrow robotRoom = ROOM0$
$robotLocPar = LOC00 \vee robotLocPar = LOC01 \Rightarrow \ robotRoomPar = ROOM0$ etc.

CSP constraints, $\forall_{0 \leq i < k}$:
`prec constraints:`
$moveRobot[i] = true \Rightarrow$
$robotLocPar\_known[i] = true \ \wedge \ robotRoomPar\_known[i] = true \ \wedge \ robotLoc = robotLocPar \wedge$
$((robotLoc[i] = LOC00 \wedge robotLocPar[i] = LOC01) \vee$
$(robotLoc[i] = LOC01 \wedge robotLocPar[i] = LOC00) \vee \ldots)$
$\vee \ ((robotLoc[i] = LOC00 \wedge robotLocPar[i] = LOC10 \ \wedge \ doorROOM0\_ROOM1[i] = OPEN) \vee$
$(robotLoc[i] = LOC10 \wedge robotLocPar[i] = LOC00 \ \wedge \ doorROOM0\_ROOM1[i] = OPEN) \ \vee \ \ldots)$
`effect constraints:`
$moveRobot[i] = true \Rightarrow robotLoc[i+1] = robotLocPar[i] \ \wedge \ robotLoc\_known[i+1] = true \wedge$
$robotLoc\_changed[i+1] = true$

CSP-level general constraints, $\forall_{0 \leq i \leq k}$:
$robotLoc[i] = LOC00 \ \wedge \ robotLoc[i] = LOC01 \Rightarrow \ robotRoom[i] = ROOM0 \ \wedge \ robotRoom[i]\_known = true$

Frame axiom constraints guaranteeing that variables cannot change between subsequent states, unless an action that affects them takes place, are added.

For every $v \in (V - Rv)$ and for each $0 \leq i \leq k-1$, we add

$$\bigwedge_j (actionAff(v)_j = 0) \Rightarrow v[i] = v[i+1]$$

where $actionAff(v)_j$ are the actions affecting v, i.e., the actions for which v appears in the left side of an equality involved in the constraints derived from their effects.

If v appears in the right side of the implication of a general constraint, then actions whose effects involve the variable on the left side of the respective general constraint are also included in $actionAff(v)_j$. The set of constraints that comprise *CSP* are further extended by additional constraints that constitute the goal, yielding the planning problem in the form of a CSP that are passed to the constraint solver. The handling of the sensing effects allows the offline solver to assign arbitrary values to unknown variables, however, if the corresponding knowledge variable *var_known* is false, this value is of no validity. This way the planner always generates an *optimistic* plan, i.e., anticipating that all knowledge-gathering actions return information that is in accordance with the user's requirements.

2.2.1. Implicit predicates in the knowledge base

Although knowledge-level variables reflect whether a state variable is known or not, they cannot capture the presence or absence of information about functions. The question is how to model the fact that the planner knows that. E.g., a hotel room has been booked for a specific location and room parameters, in other words that *bookedHotel* is true for some values of *hPlacePar* and *hDatePar*. Grounding the domain is not a feasible option because of the many input parameters which range over large domains. Therefore, a separate modeling is required, to allow distinguishing between information that refers to input parameters, and enables the planner to make the appropriate output-to-input assignments. As new observations are made at execution time, the knowledge base facts and the constraints change.

The planner maintains two structures to store its knowledge about the values of variables. *knowlBase* is a map that keeps the values of variables predicated on certain parameter values, i.e., the fact that $var = val$, where $var \in Var$ and $val \in D^{var}$, given a set $\{(p_1 = c_1), \ldots, (p_n = c_n)\}$, where $p_i \in Par$ and $c_i \in D^{p_i}$. *knownVars* stores the known values of variables which do not depend on any parameters.

For each entry of *knowlBase* $\{(p_1 = c_1), \ldots, (p_n = c_n)\} \mapsto (var, val)$, a *virtual KB action* is added to the planning domain. This virtual action has the list $\{p_1, \ldots, p_n\}$, preconditions $\bigwedge_i (p_i = c_i)$, and an effect called virtual assign $virtAssign(var, val)$ as input parameters. The constraints capturing this assignment are the same as the ones of the standard $assign(var, val)$, except that the change denoting action *var_changed* is not set to true. Thus, a virtual KB action simulates a sensing action whose output is known in advance. In this way, grounding is performed only with respect to what the planner knows, which is expected to be limited, especially in comparison with the number of all possible configurations that could exist. Virtual KB actions are considered in the formation of frame axioms just like other actions. The planner will always try virtual actions before actual actions.

Regarding the *knownVars* list, for each $(var = val) \in knownVars$, the CSP variable $var[0]$ is assigned to *val*, and $var\_known[0]$ to true. The planner always starts from an initial state where all variables which are not part of *knownVars* are unknown. This implies that the plan has to include virtual actions, to transition to a state that represents what it actually already knows. The information included in *knowlBase* and *knownVars* may be annotated by a timestamp and expiration time, after which it is removed from the map, i.e., considered not to be known anymore. Therefore, the initial state and the set of virtual KB is constructed anew every time the planner is triggered.

The CSP solver may choose any virtual action reflecting the *knowlBase* map suiting its purposes, and assign input parameters accordingly. For example, consider a goal about delivering a parcel to the address where a given person, "Peter Pan", lives. If *knowlBase* already contains entry $namePar =$ "*PeterPan*" $\mapsto$ $catalAddress =$ "*Neverland*", then the planner chooses the respective virtual action *KBSense1* with input parameters $namePar =$ "*PeterPan*" to retrieve the desired value for *catalAddress*, and then proceed to the reservation action.

2.3. Goal language

The goal language supported by the RuGPlanner provides the user with expressive constructs for stating complex goals, beyond the mere statement of properties that should hold in the final state. Conditions over state traversals, maintainability properties, and distinguishing between wish to observe the environment and wish to change it are some of the features this language supports. The goal language shares many common concerns with the aspects presented in [48], such as the distinction between hands-off observations and accomplishment goals. The goals are translated into a set of constraints which together with the constraints formulating the planning domain and initial state constitute the final CSP which is passed to the constraint solver.

Goal syntax

The goal syntax is defined in the following way:

goal	::=	$\bigwedge_i$(*condition-goal*$_i$ | *condition_or_not-goal*$_i$ | *subgoal*$_i$)
condition-goal	::=	(*subgoal*) `under_condition` *goal*
condition_or_not-goal	::=	(*subgoal*) `under_condition_or_not` *goal*
subgoal	::=	`final` (*props*) | `all_states` (*props*) | `achieve`(*props*) | `achieve-maint` (*props*) | `find_out-maint` (*props*) | `find_out` (*props*)

where *props* is a propositional formula *precond*(*a*) as defined in Definition 1, with $var, var_1, \ldots, var_n \in (Var \cup Par)$. All variables and parameters not specified in the goal (or the initial state constraints) are assumed to be undefined (i.e., their respective knowledge-level variables are set to false).

The `final` subgoal is satisfied if *props* holds at the last state, while `achieve` requires that *props* should be true at *some* state over the state traversal. The `maint` annotation adds the requirement that once the respective propositions become true at some state, they should remain true in all subsequent states of the plan till the final one. `all_states` imposes that *props* should be true at all states of the plan, and is usually applied on input parameters whose values are by the user. The `find_out` type of subgoals enforces a hands-off requirement on the variables the respective propositions involve, i.e., the planner tries to satisfy the propositions at some state without allowing any world-altering effect on these variables before that state. `find_out-maint` extends the hands-off requirement on the involved variables till the final state, i.e., they should remain intact at all states of the plan. For instance, the goal `find_out`(*account_balance* > *100*) will be satisfied if the sensed value of the variable *account_balance* is greater than 100, without however allowing any action to alter the variable's value before the sensing action. On the other hand, if the goal is `achieve`(*account_balance* > *100*), the planner will do everything possible in order to fulfill the proposition, e.g., it might invoke a *pay_in* action that increases the *account_balance* by some amount.

Subgoals can be further on combined through the condition goal constructs, which impose conditions that should be assured before the fulfillment of the subsequent subgoal. *subgoal*$_0$ `under_condition` *goal*$_1$ is satisfied if *subgoal*$_0$ is satisfied for the first time at some state *s* (see Definition 7) and *goal*$_i$ is satisfied in the state sequence preceding *s*. `under_condition` thus imposes a before-then relation between goals over the state traversal, and is particularly useful in cases where the user would like to go ahead with altering a variable, only if its sensed value satisfies a property beforehand. `under_condition_or_not` allows the expression of what can be seen as a sort of soft requirements: *subgoal*$_0$ `under_condition_or_not` *goal*$_1$ will also be fulfilled if *goal*$_1$ is not satisfiable; if however it is, then *subgoal*$_0$ has to be as well.

2.3.1. Goal examples

Next, we present two simple examples to demonstrate the use of the goal language. More examples of goals that express the requirements of different scenarios can be found in [68,114].

Example 2

Goal 1
`achieve-maint`(*bookedConcert* = *TRUE*) `under_condition` (`find_out-maint`(*temperature* > *0*))

Goal 2
`achieve-maint`(*bookedHotel* = *TRUE*) ∧ (`achieve-maint`(*bookedConcert* = *TRUE*) `under_condition_or_not` (`find_out-maint`(*temperature* > *0*)))

Goal 3
$\bigwedge_i$`achieve`(*robotLocation* = *room*$_i$)

Goal 1 is accomplished if *s* is the first state at which *bookedConcert* = *TRUE* is satisfied, and `find_out-maint` (*temperature* > *0*) holds in the state sequence preceding *s* (in this example, the maintainability requirement imposed by `find_out-maint` is in practice redundant because there is no way to change the weather). If *temperature* < *0*, Goal 1

fails. On the other hand, Goal 2 ensures that *bookedConcert* = *TRUE* will be satisfied if the temperature is not below zero, while if it is, then only *bookedHotel* = *TRUE* will be looked after. Goal 3 states that a robot should visit *all* rooms in a house, leaving the order of visits to be computed by the planner, depending on the house structure.

2.3.2. Goals with parameters

We now face the problem of representing functions in the goal, e.g., how to say that *bookedHotel*(*hPlacePar*, *hDatePar*) is desired, where *hPlacePar* and *hDatePar* can be either a specific value or refer to another variable. In approaches where actions and propositional functions are grounded, in order to reach a final state that satisfies *bookedHotel*("*Groningen*", "*12-02-2014*"), the respective propositional variable should appear in the effects of an operator's grounded instance (e.g., *bookHotel_Groningen_12022014*). But how can such a goal be expressed and satisfied in a variable-based ungrounded context? Actually, what the expression *bookedHotel*("*Groningen*", "*12-02-2014*") implies is that the input arguments of the action that fulfills *bookedHotel* should be set to "*Groningen*", and "*12-02-2014*" respectively. The effects of this action satisfy the proposition by setting the variable *bookedHotel* to true.

To capture such expressions we introduce the notation *prop* `withParams` $\bigwedge_j par_j = v_j$ where *prop* refers to a proposition that should hold at state i, and $\bigwedge_j par_j = v_j$ with $par_j \in Par$ should hold at state $i-1$. v_j can be either a constant $v_j \in D^{par_j}$ or a variable $v_j \in Var$. According to this notation we would thus write as in Example 3.

Example 3 A goal that requests booking two hotel rooms on different dates looks like:

`achieve`(*bookedHotel* = *true* `withParams` (*hPlacePar* = "*Groningen*" ∧ *hDatePar* = "*12-02-2014*")) ∧
`achieve`(*bookedHotel* = *true* `withParams` (*hPlacePar* = "*Rotterdam*" ∧ *hDatePar* = "*13-02-2014*")).

Output-to-input matchings can be captured in the goal by binding parameters to other variables. Recalling the example in Section 2.2.1, the goal would look like this:

`final`(*delivery* = *true* `withParams` (*destinationPar* = *catalAddress*))
`under_condition` (`find_out-maint`(*known*(*catalAddress*) `withParams` (*namePar* = "*PeterPan*")))

The second goal in Example 3 expresses the request to deliver to the address of "Peter Pan", which can be retrieved for example from a sensing action provided by an online catalog. That sensing action should be performed with the right assignment (*namePar* = "*PeterPan*"), and the delivery should be performed on the respective output, otherwise the goal cannot be satisfied (see also the concrete semantics of the goal constructs in Section 2.4).

If the knowledge base already includes the entries (*namePar* = "*PeterPan*") ↦ (*catalAddress* = "*Neverland*"), and (*namePar* = "*Alice*") ↦ (*catalAddress* = "*Wonderland*"), then the following two virtual KB actions are added to the planning domain, as described in Section 2.2.1:

KB1(*namePar*)
prec: *namePar* = "*PeterPan*"
eff: *virtAssign*(*catalAddress*, "*Neverland*")

KB1(*namePar*)
prec: *namePar* = "*Alice*"
eff: *virtAssign*(*catalAddress*, "*Wonderland*")

Given these facts, the planner produces the plan: *KB1*(*namePar* = "*PeterPan*"), *deliver*(*destinationPar* = "*Neverland*"). A goal which requests a delivery to both Alice and Peter Pan is also satisfiable, by a plan like:

KB1("*PeterPan*"), *deliver*("*Neverland*"), *KB2*("*Alice*"), *deliver*("*Wonderland*")

(for readability reasons, we put directly the input parameters values along with the actions). The applicability of *knowlBase* becomes more evident in Section 4, where the entries in the knowledge base change according to the observations made during execution.

2.4. Representing the planning problem

Based on the planning domain as described in Definition 1, a *State Transition System* (STS) Σ evolves by specifying a *state-transition function* γ on the state and action sets. γ is applied to a state and leads to a *set* of states. This is due to the fact that *effects*(a) do not only model assignments to values, but also outcomes that are unknown offline. As described in the process of constraints derivation in Section 2.2, *effects*(a) entail a conjunction of propositions $constr\_effect_j$ that should hold at the successor state. Recalling that a state s satisfies a formula *props* only if all possible combinations of values that are members of the domains of the variables at s satisfy *props*, it follows that an effect of type *sense*(*var*) leads to a set of states: the proposition *var* = *var_response* should hold at the successor state, with *var_response* $\in D^{var}$, which amounts to $|D^{var}|$ different states. This way, the function γ captures incomplete knowledge and offline non-determinism in the case of knowledge-gathering actions. To support the possibility of applying multiple concurrent actions at a single transition, and to be able to give a definition of plan including parallel actions, the function γ takes a set of actions as argument.

Definition 4 *(State transition system).* A state transition system based on a planning domain extended with the knowledge-level representation $\mathcal{PD}' = \langle V, Par, Act \rangle$ is a triple $\Sigma = \langle S, Act, \gamma \rangle$, such that:

- S is a set of states $S = \{s = \{(x_1, D_s^{x_1}), \ldots, (x_n, D_s^{x_n})\}$, with $x \in V \cup Par$ and $D_s^{x_i} \subseteq D^{x_i}\}$.
- Act is the set of all actions in PD.
- $\gamma : S \times \wp(Act) \rightarrow \wp(S)$ (where $\wp$ denotes the powerset) is a transition function such that given a state s and a set of actions $A = \{a_1, \ldots, a_n\} \subseteq Act$ such that $precond(a_i)$ hold at s for all $a_i \in A$, then the application of all $effects(a_i)$ on s leads to a set of successor states S_s. If for some a_i, $precond(a_i)$ do not hold at s, or if $A = \emptyset$, then $\gamma(s, A) = \emptyset$.

By generalizing on sets of states S, we define $\hat{\Gamma}(S, A) = \bigcup_{s \in S} \gamma(s, A)$. We can now proceed to the definition of the planning problem and plan.

Definition 5 *(Planning problem).* A *planning problem* is a triple $P = \langle \Sigma, S_0, g \rangle$, where Σ is a transition system as in Definition 4, S_0 is the set of all states which satisfy a conjunction of propositions $prop\_init_i$, and g is a goal.

Definition 6 *(Plan).* A *plan* consists of a sequence of action sets $\pi = \langle A_0, \ldots, A_{k-1} \rangle$, where k is the length of the plan, and a sequence *inPars* of assignment relations $inPars_i$ for each $A_i \in \pi$. $inPars_i$ is defined as $\{(p := c_p) \mid \forall p \in in(a)\ \forall a \in A_i\}$, where $c_p \in D^p$.

A plan which instructs sensing the balance of an account first, and then increasing it by a certain amount (in line with the actions of Example 1) is represented in Example 4.

Example 4
$\pi = \langle \{findAccBalance\}, \{payIn\} \rangle$
$inPars = \langle \{accIdPar1 := 14\,382\}, \{amountPar2 := 1000, accIdPar2 := 14\,382\} \rangle$

We extend the function $\hat{\Gamma}$ to capture the set of states that are brought forth by applying the actions in π, starting from $S_0[inPars_0]$, where $S_0[inPars_0]$ is S_0 with the domains of input parameters updated according to $inPars_0$. Given an action sequence $\pi = \langle A_0, \ldots, A_{k-1} \rangle$, and an $inPars = \langle inPars_0, \ldots, inPars_{k-1} \rangle$ we use the notation: $\Gamma(S) = \hat{\Gamma}(S[inPars_0], A_0)$, $\Gamma^2(S) = \hat{\Gamma}(\Gamma(S)[inPars_1], A_1)$, and similarly for $\Gamma^3(S), \ldots, \Gamma^k(S)$. Thus, a plan consisting of π and *inPars* imposes a sequence of state sets $\tilde{S} = \langle S_0, \Gamma(S_0), \Gamma^2(S_0), \ldots, \Gamma^k(S_0) \rangle$. We call $\tilde{S}$ the *execution path* computed offline. Note that the transition function is applied on a subset of the state set resulting from the previous transition, as induced by the sequence of input parameter assignments in the plan. In the next section, we formally describe when a plan π has the potential to solve the planning problem P, i.e., when the application of π yields an $\tilde{S}$ that satisfies the goal g.

2.4.1. Semantics of the goal

The notion of goal satisfaction is defined in terms of the execution path $\tilde{S} = \langle S_0[inPars_0], \ldots, S_k \rangle$ induced by a planning problem $P = \langle \Sigma, S_0, g \rangle$, input parameters assignment *inPars*, and a sequence of action sets $\pi = \langle A^0, \ldots, A^{k-1} \rangle$. We will use the notation $S \supseteq props$ if there is at least one state $s \in S$ that satisfies the propositional formula *props*. We say that a plan has the potential to solve the planning problem, if it corresponds to an execution path which subsumes a sequence of states that satisfy the propositions inferred by the goal. We denote the index of the last state set in an execution path with $last(\tilde{S})$. We first introduce the notion of the minimal execution path.

Definition 7 *(Minimal execution path).* $min(\tilde{S}, props) = \langle S_0, \ldots, S_n \rangle$ is a subsequence of $\tilde{S}$, so that $(S_n \supseteq props) \wedge (\forall i,\ 0 \leq i \leq n-1 : \neg(S_i \supseteq props))$. Thus, $min(\tilde{S}, props)$ represents the execution path whose final state set S_n is the first one in the sequence that contains a state that satisfies *props*.

For example, given the execution path $\langle S_0, S_1, S_2 \rangle$ imposed by the plan of Example 4, the minimal execution path which satisfies the proposition $accBalance\_known = true$ is $\langle S_0, S_1 \rangle$, since the state S_1 is the first state in which the proposition holds.

We say that an execution path $\tilde{S} = \langle S_0, \Gamma(S_0), \ldots, \Gamma^k(S_0) \rangle$ *has the potential to solve* the planning problem P given a set of initial states S_0 and a goal g, and we write $\tilde{S} \models g$ if:

$\tilde{S} \models \texttt{final}(props) : S_k \supseteq props\_and\_known$ where $props\_and\_known = props \wedge \bigwedge_{var_i \in props} var_i\_known = true$

$\tilde{S} \models \texttt{all\_states}(props) : \forall S_j \in \tilde{S} : S_j \supseteq props\_and\_known$

$\tilde{S} \models \texttt{achieve}(props) : \exists S_j \in \tilde{S}$ such that $S_j \supseteq props\_and\_known$

$\tilde{S} \models \texttt{achieve-maint}(props) : \tilde{S} \models \texttt{achieve}(props) \wedge$
$\forall j,\ k \geq j \geq last(min(\tilde{S}, props\_and\_known)) : S_j \supseteq props\_and\_known$

$\tilde{S} \models \texttt{find\_out}(props)$: $\tilde{S} \models \texttt{achieve}(props \wedge \bigwedge_{var_i \in props \text{ and appear in world-altering effects}} var_i\_changed = false)$

$\tilde{S} \models \texttt{find\_out} - \texttt{maint}(props)$: $\tilde{S} \models \texttt{find\_out}(props) \wedge \forall j,\ k \geq j \geq last(min(\tilde{S}, props\_and\_known)) : S_j \supseteq (props \wedge \bigwedge_{var_i \in props \text{ and appear inworld-altering effects}} var_i\_changed = false)$

$\tilde{S} \models sg$ `under_condition` $goal$: $\tilde{S} \models sg \wedge min(\tilde{S}, props\_and\_known(sg)) \models goal$
where $props\_and\_known(sg)$ are the propositions corresponding to sg plus the requirement that all variables involved in them are known

$\tilde{S} \models sg$ `under_condition_or_not` $goal$: $(\tilde{S} \models goal) \Rightarrow \tilde{S} \models (sg$ `under_condition` $goal)$
$\wedge \exists S_j \in \tilde{S} : S_j \supseteq known(goal)$ where $known(goal)$ means $\bigwedge_{var_i \in props(goal)} var_i\_known = true$

$\tilde{S} \models \bigwedge_i goal_i$: $\bigwedge_i (\tilde{S} \models goal_i)$

As in the case of the constraints entailed by the preconditions presented in Section 2.2, an extra requirement that all variables involved in *props* should be known is added. This implies that, setting aside uncertainty stemming from sensing effects, a plan has the potential to solve the planning problem if the goal is satisfied for all possible assignments to variables allowed by *prop_init*. Thus, given some *prop_init* that imply $(1 \leq v \leq 2)$, an empty action set and the goal $g = \texttt{final}(v = 1)$, g is not satisfiable. If, on the other hand, there is a sensing action with effect $sense(v)$, there is a plan that has the potential to satisfy the goal. However, this extra requirement that all variables in the goal should be known may exclude plans that would otherwise be considered acceptable. For example, given the same *prop_init* and an empty action set, the goal $\texttt{final}(v < 3)$ is also not satisfiable, despite the fact that it holds for all possible assignments to v. This strong restriction is necessary to prevent the constraint solver from presenting trivial assignments as acceptable solutions. Thus, the term *potential to solve* refers to the uncertainty of outcomes during sensing, but not to the uncertainty due to the incomplete knowledge about the initial state.

The goal is translated into a set of constraints on the CSP-level state variables, which are added to the set of constraints formulating the planning domain. The details of this translation process can be found in [65]. A web-based graphical goal editor, which is designed to assist the user in specifying an extended declarative goal given a planning domain has been implemented in [125].

3. Planning by solving the CSP

The set of constraints resulting from the translation of the planning domain, the propositions referring to the initial state and the goal form the CSP are passed to the constraint solver. The constraint solver computes a valid assignment to the CSP variables that model the planning actions, and this assignment corresponds to an optimistic plan that has the potential to solve the planning problem.

3.1. Solving the CSP

Prior to calling the solver, the planner prunes from its search space the actions about which it knows in advance they have no potential to contribute to the goal satisfaction, in a fashion similar to [94]. This preliminary process identifies all actions a_i that include at least one of the goal variables in their effects, and then recursively finds all actions which include in their effects variables that are involved in the preconditions of the actions a_i that are directly related to the goal. The search for applicable actions during solving is thus limited to this set of possible candidates, an effect that may considerably facilitate the solver's work in situations where there are many actions available. Along with this preliminary pruning, a value selection strategy that first tries to assign false values to the action variables is employed. In this way, the inclusion of redundant sensing or even unwanted world-altering actions in the produced plans is usually avoided. Yet, it does not guarantee that the computed plans are optimal, i.e., that they include the least possible number of actions which fulfill the goal.

The solving process proceeds through a combination of consistency techniques and search (branching) algorithms. Constraints are propagated using the GAC3rm algorithm [82]. Usually, search strategies that yield good-quality plans have worse performance than strategies which lead to plans that include redundant actions, e.g., by applying a random value assignment. In the remainder of this work, a "most constrained" variable selection heuristic and an "increasing domain" value iteration strategy is employed in the testing process, unless stated otherwise. Most constrained implies selecting the variable involved in the largest number of constraints. Variables modeling virtual KB actions are selected before all others. Then an iteration over values in increasing order takes place.

Regarding the choice of k, this is selected depending on the planning domain. It could be restricted by the number of grounded action instances, however since this can be very high (given the potentially large domains of the input parameters), k is set by the domain designer, based on the maximum size of expected plans. For example, given a domain, where a robot has to move between locations, k could be set to 3 times the number of locations. Due to the high degree of parallelism that characterizes the produced plans, many solutions which require considerably more than k actions will be found.

3.2. A planning example

Let us now consider a planning problem which models the scenario described in Section 1.3. The planning operators simulate the functionality of services that reside on the Web, and provide information about entertainment events, maps, calendar, weather, and hotels as well as offering the possibility for booking concert tickets and hotel rooms. The actions can be mapped to the APIs of real services, as we showed in [67]. Let us consider a user who lives in Groningen and wants to book a ticket and a hotel room for the nearest upcoming concert of the band "Neutral Milk Hotel" whose date and location meet criteria about weather conditions, the distance from Groningen, his availability on the performance day according to his agenda, as well as about the price he is willing to pay for his overnight stay. This wish is captured by the nested goal presented in Example 5.

Example 5 *Entertainment goal*

achieve-maint((*bookedHotel* = *TRUE* ∧ *hotelPrice* < *80*) withParams
(*hPlacePar* = *eventPlace* ∧ *hDatePar* = *eventDate* ∧ *numbOfNightsPar* = *1* ∧ *roomTypePar* = "*single*"))
under_condition
achieve-maint((*bookedTicket* = *TRUE*) withParams
(*bandNamePar* = "*NeutralMilkHotel*" ∧ *concDatePar* = *eventDate*))
under_condition
find_out((*temperature* > *0*) withParams (*wPlacePar* = *eventPlace* ∧ *wDatePar* = *eventDate*)
∧ (*busy* = *FALSE*) withParams (*cDatePar* = *eventDate*)
∧ (*distance* < *200*) withParams (*mapsOriginPar* = "*Groningen, NL*" ∧ *mapsDestinPar* = *eventPlace*))

The variables *eventPlace* and *eventDate* on which the performance will take place are unknown offline, and it is up to some knowledge gathering service (namely the eventful.com service) to provide them. In the initial optimistic plan these are assigned some convenient value by the solver, however, the respective knowledge-level variables indicate that this value is not a valid one. An assignment to a variable *var* = *value* for which *var_known* = *false* is signaled in the optimistic plan by a "defaultVar" mark.

By employing the conservative combination of most constrained and increasing domain selection strategies, the plan is generated offline for the entertainment goal, Example 6.

Example 6

{*getEventsList*(Neutral Milk Hotel)},
{*getNextEvent*},
{*checkCalendarAvail*(defaultDate),
getDistance(Groningen, defaultPlace),
getTemperature(defaultPlace, defaultDate),
getAvailHotels(defaultDate, defaultPlace, 1, 1)},
{*bookConcertTicket*(Neutral Milk Hotel, defaultDate)},
{getNextHotelInfo},
{*bookHotel*(defaultHotel, defaultDate, 1, 1)}

For readability reasons, we put the assignment to the input parameters together with the actions. Actions within curly brackets correspond to a set of actions which are applied in parallel. The values "defaultPlace", "defaultDate" and "defaultHotel" all correspond to the same values, i.e., to the yet unknown *eventPlace*, *eventDate* and *hotelId* sensed by *getNextEvent* and *getNextHotelInfo*. *getEventsList* computes the list of performances for a given band ordered by date. The service for dealing with hotels provides aggregated searching and booking facilities over a wide range of hotel providers (like services as booking.com do), and orders the results according to some criteria (e.g., price). *getNextHotelInfo* returns the information (price, hotel id) of the next hotel in the formed list.

4. Plan orchestration via alteration of the CSP

The suitability of the RuGPlanner for dynamic service environments lies in the idea of delaying the computation of alternative plans until this is called for by the new information acquired during execution. Continual planning is performed, so that the upcoming plan steps anticipated offline can be revised as execution progresses, in face of inconsistencies that arise either from the newly acquired information, from services' inconsistent behaviors or from the actions of exogenous

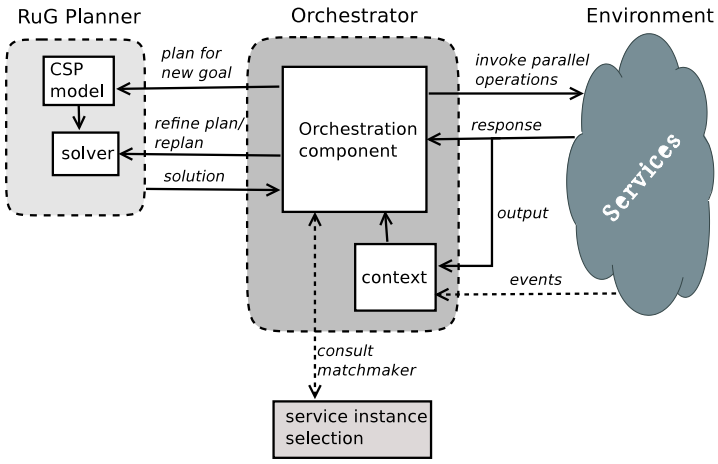

Fig. 1. Architectural overview of the orchestrator.

agents that interfere with the plan. In such a setting, the goal is considered to have been accomplished, if all individual actions in this sequence of updated plans are successfully executed. We call the process that starts from an initial plan and moves on by interweaving action invocations with plan revisions an *orchestration*. Orchestration is performed by applying gradual modifications to the CSP instance which models the planning domain, the goal and the constantly changing contextual state. The modifications correspond to the incorporation of new facts about the state of the world or the removal of obsolete ones, to refinements of the sequence of actions included in the already computed plan, or to useful information about the behavior of services that is collected by inspecting how they operate. The orchestration algorithm is characterized by the following traits:

- It exploits the high degree of parallelism in the plan, by performing concurrent invocations and handling the responses in a non-blocking way. Since the execution time of some service operations may be very long, the orchestrator is able to continue planning and proceed to the execution of subsequent actions if this is allowed by the domain and goal restrictions, while waiting for the response of a service invocation.
- It provides the means to recover from failure responses and timeouts. Other arbitrary service outcomes that contradict its expected behavior can also be tolerated under the assumption of a consistent and timely publish-subscribe mechanism.
- It can cope with possible discrepancies due to the activity of exogenous agents, which may act in parallel with the plan execution and interfere with it.
- It seeks to keep a balance between the effort spent in computing a new plan from scratch and in refining an existing one.
- It takes care of the data flow by instantiating numeric-valued input parameters to the actually sensed output parameters. This is performed through plan refinement, by considering the history of known facts in the form of constraints and the goal requirements.

4.1. Architectural overview

The architecture that realizes the interweaving between synthesis and execution-time coordination is presented in Fig. 1. Whenever a request about computing a plan for a new goal is issued, the orchestrator asks the planner to compute an initial optimistic plan. This entails adding dynamically the constraints that follow from the goal to the model kept by the RuGPlanner, i.e., the constraint network that models the planning domain. The *solver* takes into consideration the current model along with some assignments to CSP-level variables that reflect the initial planning state, as delivered by the current context. The solution to the CSP which amounts to an offline plan (if one exists) is passed to the orchestrator, whose task is to gradually execute and update it, according to the information it acquires through its interaction with the environment. Every step of the plan involves a set of parallel actions, which are mapped to a set of respective concrete service operations executed concurrently.

Requests for refinement are addressed directly at the solver level of the RuGPlanner, as long as the basic model remains the same (i.e., changes refer only to the initial state). In this way, the search process can start from an already propagated instance of the model, and proceed from a state where some variables are already instantiated according to the most up-to-date context.

Dashed arrows in Fig. 1 correspond to interactions that are only available under certain assumptions. Change events can be received asynchronously if this is supported by an underlying publish/subscribe mechanism (as, e.g., in the OSGi framework [68]). The instantiation of an action to the operation offered by a specific service provider is taken care by an

external component, which is responsible for discovering and selecting the appropriate service instances. The process of service discovery and selection is an interesting and difficult problem by itself, e.g., see [109,101], and is out of the scope of the current work.

4.2. Execution-time transition system and orchestration path

Recalling Definition 4, the STS Σ has to be extended in order to capture observations and external events. For simplicity, we assume that: (i) if an action is applied, all of its effects take place as prescribed, (ii) observations and events refer to disjoint sets of variables, and (iii) all observations corresponding to a set of actions are retrieved timely, i.e., before the application of the next sets of actions in a plan. Failed actions and byzantine behaviors can be modeled indirectly, by introducing events which assign certain variables after the application of an action.

Definition 8 *(Execution-level state transition system).* An execution-level state transition system based on a planning domain $\mathcal{PD}' = \langle V,\ Par,\ Act\rangle$ (where $V = Var \cup Kb \cup Cv \cup Rv$) is a tuple $\Sigma_e = \langle S, Act, Ev, Obv, \zeta\rangle$, where:

- S is a set of states $S = \{s = \{(x_1, D_s^{x_1}), \ldots, (x_n, D_s^{x_n})\}$, with $x \in V \cup Par$ and $D_s^{x_i} \subseteq D^{x_i}\}$.
- Act is a set of actions.
- Ev is a set of events. An event is an assignment to some variable ($var := val$), where $var \in Var$ does not participate in any observational effect, and $val \in D^{var_i}$.
- Obv is a set of observations. An observation is an assignment to a response variable ($var\_response =: val$), where $var\_response \in Rv$ and $val \in D^{var}$.
- $\zeta : S \times \wp(Act) \times \wp(Obv) \times \wp(Ev) \rightarrow \wp(S)$ is the execution-level state transition function $\zeta(s, A, O, E) = \{\delta(s', O, E) \mid \forall s' \in \gamma(s, A)\}$, where $A \subset Act$, $O \subset Obv$, $E \subset Ev$, and:
  - all assignments in E and O refer to different variables.
  - γ is defined in Definition 4.
  - $\delta : S \times \wp(Obv) \times \wp(Ev) \rightarrow S$ is a function which updates a state s by applying the assignments in E and O.
  - For every var which appears in a sensing effect $sense(var)$ of some $a_i \in A$, $var\_response$ is part of an observation $o_i \in O$.

Generalizing on sets of states S, we define: $\hat{Z}(S, A, O, E) = \bigcup_{s \in S} \zeta(s, A, O, E)$.

Definition 9 *(Orchestration problem).* An *orchestration problem* is a triple $OP = \langle \Sigma_e, S_0, g\rangle$, where Σ_e is an execution-level state transition system, S_0 is the set of all states that satisfy a conjunction of propositions $\bigwedge_i prop\_init\_i$, and g is a goal as specified in Section 2.3.

Definition 10 *(Orchestration).* An *orchestration* is a sequence of triples of sets of actions, events and observations $orch = \langle (A_0, O_0, E_0), \ldots, (A_n, O_n, E_n)\rangle$, and a sequence $inPars$ of assignment relations $inPars_i$ as defined in Definition 6 for each A_i that appears in $orch$.

In an orchestration, the sequence of events and observations is uncontrollable, and the sequence of actions and input parameters is selected by the planner. We call the sequence of actions $\langle A_0, \ldots, A_n\rangle$ in $orch$ along with $inPars$ the execution-level plan π_e. Similarly to Section 2.4, we extend the $\hat{Z}$ function to capture the sequence of set of states that are brought forth by $orch$ and $inPars$, starting from S_0. Given an orchestration

$$orch = \langle (A_0, O_0, E_0), \ldots, (A_n, O_n, E_n)\rangle,$$

we use the notation: $Z(S) = \hat{Z}(S[inPars_0], A_0, O_0, E_0)$, $Z^2(S) = \hat{Z}(Z(S)[inPars_1], A_1, O_1, E_1)$ etc. Thus, an orchestration comprising π_e induces a sequence of state sets

$$\tilde{S_e} = \langle S_0,\ Z(S_0), Z^2(S_0), \ldots, Z^n(S_0)\rangle.$$

We call $\tilde{S_e}$ the *orchestration path* or run. An orchestration path $\tilde{S_e} = \langle S_0, Z(S_0), \ldots, Z^n(S_0)\rangle$ is a solution to the orchestration problem $OP = \langle \Sigma_e, S_0, g\rangle$, and we write $\tilde{S_e} \models g$, if $\tilde{S_e}$ satisfies the properties described in Section 2.4. We say that an execution-level plan π_e is a *weak* or *optimistic* solution to the orchestration problem OP, if there is some sequence $\{(O_0, E_0), \ldots, (O_n, E_n)\}$ where $E_0 = \ldots = E_n = \emptyset$, which leads to an orchestration path that is a solution. That is, if no events occur during the orchestration and there is some convenient assignment to response variables that satisfies the goal. We say that π_e is a *strong* plan, if it leads to an orchestration path that is a solution for any sequence $\{(O_0, E_0), \ldots, (O_n, E_n)\}$. Since the sequence of events and observations is unknown at planning time, we cannot say whether a plan is a solution or not before all transitions actually take place. Strong plans, i.e., plans that are a solution no matter how the execution behaves, do not exist in the systems we are interested in, since any event may occur at any transition point.

Due to the uncontrollable nature of events, dead-ends cannot be avoided, i.e., the orchestration may bring the world to a state from which the goal is no longer satisfiable. Given a partially executed plan $\{A_0, \ldots, A_{i-1}\}$, considering the current

set of states S_i, an event e_i may lead to a set of states $S'_i = \{\delta(s, \emptyset, \{e_i\}) \mid \forall s \in S_i\}$, from which no plan $\{A_i, \ldots, A_n\}$ that is an optimistic solution to the problem $OP = \langle \Sigma_e, S'_i, g \rangle$ can be computed.

5. Orchestration properties and policies

Next, we describe the properties and main policies of the orchestration algorithm. The algorithm constructs an execution plan incrementally, by taking the first set of actions of each offline optimistic plan computed by the RuGPlanner. Each optimistic plan is computed considering the states that result from the application of the δ function at each step. At each revision step, the planner considers a new CSP instance following from the current version of the knowledge base, which incorporates the information included in the latest sets of observations and events. The behavior of the orchestrator, which encompasses the following policies, is presented in pseudocode in Appendix A.

5.1. Soundness and completeness of the orchestration algorithm

Soundness. The orchestration algorithm of Appendix A is sound for goals that respect certain properties concerning soft goals and maintainability goals with variables affected by observations or events, as illustrated by Theorem 1.

Theorem 1. *The orchestration algorithm is sound, if the goal does not include (i) maintainability goals which involve variables affected by events or observations and (ii)* `under_condition_or_not` *goals.*

Proof. We prove that the execution-level plan $\pi_e = \langle A_0, \ldots, A_n \rangle$ computed by the orchestration algorithm corresponds to an orchestration $orch = \langle (A_0, O_0, E_0), \ldots, (A_n, O_n, E_n) \rangle$ which is a solution to the orchestration problem $OP = \langle \Sigma_e, S_0, g \rangle$. The algorithm proceeds by solving a sequence of planning problems $\langle P_0, \ldots, P_k \rangle$, one at each round of the replanning/plan update phase of the orchestrator, and executing a respective sequence of plan prefixes: $\Pi = \langle pre(\pi_0), \ldots, pre(\pi_{k-1}), \pi_k \rangle$, where π_i is the solution to P_i, and $pre(\pi_i)$ the prefix of π_i executed before a new plan is computed. Since $\pi_k = \langle A_k^0, \ldots, A_k^m \rangle$ is the last plan computed by the algorithm, this means that the orchestration path $\tilde{S_e^k} = \langle S_k^0, \hat{Z}(S_k^0, A_k^0, O_k^0, E_k^0), \ldots, \hat{Z}(S_k^m, A_k^m, O_k^m, E_k^m) \rangle \models g$. If $\tilde{S_e^i}$ is the orchestration path corresponding to $pre(\pi_i)$, we need to prove that $\tilde{S_e} = \langle \tilde{S_e^0}, \ldots, \tilde{S_e^k} \rangle \models g$. This holds because of Lemma 1. □

Lemma 1. *Let $\tilde{S_0}, \tilde{S_1}$ be two sequences of state sets, with $\tilde{S_0} \models g$ and $\tilde{S_1} \models g$, where g is a goal respecting the assumptions of Theorem 1. Let $pre(\tilde{S_0})$ be a prefix of $\tilde{S_0}$, whose last state is the initial state of $\tilde{S_1}$. Then the sequence $\tilde{S} = \langle pre(\tilde{S_0}), \tilde{S_1} \rangle \models g$.*

Proof. Let us define $\hat{\Delta}(S, O, E) = \bigcup_{s \in S} \delta(s, O, E)$. Let $pre(\tilde{S_0}) = \langle S_0, S_1 = \hat{\Gamma}(S_0, A_0), S'_1 = \hat{\Delta}(S_1, O_0, E_0), \ldots, S_k = \hat{\Gamma}(S_{k-1}, A_k) \rangle$ and $S'_k = \hat{\Delta}(S_k, O_k, E_k)$ be the first state of $\tilde{S_1}$. We perform a case analysis on all types of goals and their semantics as described in Section 2.4.1:

$g =$ `final`$(props)$ Trivial since $\tilde{S_1} \models g$.
$g =$ `achieve`$(props)$ Trivial since $\tilde{S_1} \models g$. Similarly for `find_out`$(props)$.
$g =$ `all_states`$(props$ If $S'_k \supseteq props$, then $\tilde{S} \models g$. If $S'_k \not\supseteq props$, then $\tilde{S_1} \not\models g$, contradiction.
$g =$ `achieve-maint`$(props)$ If $S'_k \supseteq props$, then $\langle pre(\tilde{S_0}), S'_k \rangle \models g$, so $\tilde{S} \models g$. If $S'_k \not\supseteq props$ then either (i) $S_k \not\supseteq props$, i.e., $pre(\tilde{S_0}) \models g$ and thus $\tilde{S} \models g$ or (ii) $S_k \supseteq props$, which can only happen if $props$ include variables that are affected by O_k or E_k, contradiction. Similarly for `find_out-maint`$(props)$.
$g = sg$`under_condition`cg There are two cases:
(a) If $pre(\tilde{S_0}) \models g$, then let us assume, ad absurdum, that $\tilde{S} \not\models g$. This means that either (i) $\tilde{S} \not\models sg$, contradiction (since if $pre(\tilde{S_0}) \models sg$ and $\tilde{S_1} \models sg$ then $\tilde{S} = \langle pre(\tilde{S_0}, \tilde{S_1}) \rangle \models sg$ as shown for all types of sg above) or (ii) $min(\tilde{S}, props\_and\_known(sg)) \not\models cg$, contradiction, since $pre(\tilde{S_0}) \models g$.
(b) If $pre(\tilde{S_0}) \not\models g$ then either (i) $pre(\tilde{S_0}) \not\models sg$, in which case $min(\tilde{S}, props\_and\_known(sg)) = min(\tilde{S_1}, props\_and\_known(sg))$ and because $\tilde{S_1} \models g$ $\tilde{S} \models g$ or (ii) $min(pre(\tilde{S_0}), props\_and\_known(sg)) \not\models cg$ which would imply that $\tilde{S_0} \not\models g$, contradiction.

Remark. `under_condition_or_not` goal does not satisfy the lemma in case it is satisfied by $\tilde{S_0}$ through $\tilde{S_0} \not\models cg$. Then, if $\tilde{S_0}$ achieves $props(sg)$ at some state, and $\neg props(sg)$ holds in a subsequent state, $min(\tilde{S}, props\_and\_known(sg) \not\models cg$. □

If the goal includes the soft (disjunctive) `under_condition_or_not` goal or maintainability goals on variables that are affected by sensing actions or events, the algorithm may wrongly report that it fulfilled the goal, while its semantics are violated if considering the complete execution path. In order to avoid such situations, the whole execution history should be passed to the CSP for validation and solving, which, however, would result in decreasing performance as the execution path grows.

Completeness. Given the non-determinism following from observations and external events, the orchestration algorithm may be trapped in a dead-end, and is thus incomplete. Dead-ends cannot be avoided, because the planner cannot predict the actual state and the evolution of the environment, and during the orchestration any event and observation may occur. If there are alternative offline plans, there is no bias in favor of a specific plan based on a model of the execution-level and environmental behavior. In many practical situations, a slight change in the goal may entail an outlet from the dead-end. For example if the user is flexible with respect to time, a plan which chooses a date close to the user's first preference may be acceptable. The support for soft constraints and preferences in the goal in combination with replanning is an interesting line of future research. Some simple situations can be currently addressed via disjunctions which are allowed to be part of the goal.

The orchestration algorithm remains incomplete even if one assumes that there are no external events and that an action's execution leads to a set of states which satisfy the propositions entailed by the action's effects. To give an example, let us consider a goal $\texttt{final}(var = true)$, and two actions a_2, a_3 having an effect $assign(var, true)$. a_2 has a precondition $v_2 = 1$ and a_3 a precondition $v_3 = true$. Let us also assume that there is an action a_0 with no preconditions and with effects $sense(v_2)$ and $assign(vb, false)$, and another action a_1 with effect $sense(v_3)$ and precondition $vb = true$. Starting from a situation where v_2, v_3 are unknown and $vb = true$, there are two offline plans (of equal length) that have the potential to satisfy the goal: $\langle a_0, a_2 \rangle$, and $\langle a_1, a_3 \rangle$. If in the actual world, which is sensed at execution time, $v_2 = false$ and $v_3 = true$, then the execution of the first plan ends up in a dead-end, since after the execution of a_0, a_1 is not applicable anymore. Similar situations can result from settings where actions have either exclusively observational or exclusively world-altering effects. In order to avoid being trapped in dead-ends, and under the assumption that knowledge persists, it is a good practice to perform sensing actions as early as possible, and wait for their response before proceeding to world-altering actions. However, this practice may come at the cost of performance, as demonstrated in the example of Section 6.3.2.

5.2. Plan repair vs. replanning

The time spent in consistency checks, i.e., checks to establish whether the plan is still valid under the new context, and plan updates becomes dominant in the overall planning and execution time till the goal is satisfied (see, e.g., the evaluation results in [67]). Even in the most common case when an output is being sensed, the time for instantiating subsequent input arguments which depend on newly acquired information and inspecting whether this leads to a conflict may be considerable. One way to perform the necessary plan updates is to completely disregard the previous solution-plan and perform replanning from scratch (this is the approach we initially adopted [67]). Another way is to try to reuse parts of the existing plan as the building blocks for constructing a new plan, as in the strategies adopted by [35,118,115]. However, under certain circumstances the effort spent on modifying a plan can be larger than the effort required to generate a new one [95]. It should be noted that in the scenarios we are interested in, maintaining minimal perturbation or plan stability [35] of the original plan is not a concern per se. We are rather interested in computing good quality plans in short time from the current state onwards.

In the orchestration approach we adopt, we try to establish a middle-ground between investing too much effort in adjusting the current plan suffix, and directly proceeding into computing a new one from scratch. We therefore try within a time limit, usually a fraction of the time required to compute the original plan, to perform a fast search for refining the plan. In terms of the CSP representation, plan repair amounts to dealing with the dynamic CSP problem, where a CSP is subject to a sequence of alterations, i.e., additions and removals of value assignments and constraints. There are several methods which rely on CSP solution reuse to speed up the task of finding a consistent assignment to the altered CSP, e.g., [115] which exploits no-good learning. These methods, however, are beneficial under certain assumptions, e.g., when context changes correspond to constraints/value additions and deletions that pertain to a few variables (scope), or when the changes to the CSP are monotonic (relaxation through removal of a constraint or restriction through addition of a constraint).

The refining process we adopt attempts to construct within a time limit a new plan by adding extra actions at the beginning, the end or in parallel with actions of the existing plan, and allows input parameters of the actions in the current plan to take different values. In this way, output collected at the last step of execution is taken into account, so that input arguments to subsequent actions which depend on that output can be instantiated accordingly in the updated plan. At the CSP level, this approach amounts to constructing a partial assignment consisting of the action variables participating in the original plan, while leaving the rest of the variables to be assigned by the search strategy. Performing un-refinement, i.e., determining certain actions in the original plan as being redundant or even hinder the goal fulfillment given the new initial state, can be particularly difficult and time-consuming (repeatedly reserving old assignments can lead to tremendous thrashing [118]), and therefore we directly resort to replanning from scratch if no augmented plan can be found.

Every time a bundle of concurrent actions in the current plan is executed, and all respective responses are received or an average expected response time elapses, it may be necessary to check whether the remainder of the plan is still valid under the new context. Consistency checks are performed at every step if a notification mechanism is available, giving the opportunity to the planning agent to compare the expected planning state with the actual state. In such a case, the context encapsulates all the world-altering results of the services invoked by the planning agent, independently of whether these are in conformance with the expected effects or not, as well as the world-altering behavior of probable exogenous agents. Consistency inspection is very quick, since it amounts to passing to the solver a complete assignment. After the phase of the parallel execution of some actions at step i completes, the solver is passed a full assignment to variables and parameters,

which is the same as the assignment corresponding to the current plan, except that variables at state $i+1$ are assigned the values delivered by the current environmental state. If there is no notification mechanism, then all world-altering effects are materialized as prescribed in the planning domain, and no consistency check is performed. In this case, validity has to be checked only when new information is sensed.

Regarding the new information accumulated by possible sensed outputs, this is incorporated into the knowledge base *knowlBase* (see Section 2.2.1). At each plan revision, the respective virtual KB actions are extracted, and the respective constraints are added to the CSP, after removing the ones referring to the obsolete knowledge base. The constraints induced by the knowledge base ensure that the refinement process leads to the appropriate output-to-input assignments. If the plan suffix is found to be invalid with respect to the current environmental state, then an attempt to extend the plan is made. The maximum plan length is prespecified (see Section 2.2) and is the same for all planning attempts. More specifically, let us consider a plan $\pi = \langle A'_0, A'_1, \ldots, A'_{k-1} \rangle$ (see Definition 6), and define $\hat{\pi} = \langle A_0, A_1, \ldots, A_{n-1} \rangle$, $n \leq k$, as the sequence of non-empty action sets in π, respecting order. Assuming that k is always selected to be quite larger than the maximum number of plan steps, after the phase of parallel execution of A_0 completes, and depending on the collected outcomes, $\hat{\pi}$ can be augmented by adding extra actions before, after or in parallel with the actions in the suffix $\langle A_1, \ldots, A_{n-1} \rangle$. The refining process shifts the plan suffix leaving $d = \lceil (k-n)/2 \rceil$ "empty" places before it, and the rest after it. To do so, it constructs a *partial assignment* at the solver level: $\{a[d+i-1] = 1 \mid \forall a \in A_i,\ 1 \leq i < n\}$, where $a[j]$ is the CSP-level action variable representing a at state j. Input parameters are left open to be assigned by the solver, depending on the updated initial state, which includes the most up-to-date variables about the world. Any new observation (value of some *var_response*) returned after the completion of A_0 is added to *knowlBase*, predicated on the specific input parameter values with which A_0 were called. This entails a modification to the CSP model, through the addition of the respective virtual KB actions. The updated model and partial solution are passed to the solver, and the search process attempts to find an assignment for the remaining variables. If this process fails to find a valid solution within some limited time, then the instantiations to the action variables are removed, the search retracts to the generic model instance, and a new solution is sought with a partial assignment reflecting only the initial state.

5.3. Executing parallel actions and dealing with timeouts

One of the advantages of the RuGPlanner is that the produced plans are distinguished by a high degree of parallelism. This property can prove highly beneficial at execution time, especially since service compositions are likely to involve many sensing operations that can be performed in parallel. Moreover, even if only a subset of a bundle of concurrent actions complete successfully and within a time frame, this may be enough to enable the invocation of subsequent actions in the plan. This is an eager and optimistic strategy based on the assumption that fulfilling part of the goal is desirable, even if the goal as a whole is not satisfiable.

Actions which according to the domain and the goal depend on operations that are still pending are postponed till the effects of the slow operations on which these depend have been substantiated. If it turns out that there is no way (through invoking alternative service providers or by pursuing a different plan) to achieve the effects which are necessary for proceeding to the reliant actions, the orchestrator returns with an infeasibility notification, but the tasks that are independent of these unattainable effects have already been fulfilled. This eager-to-execute policy works in the following way. Considering a sequence $\hat{\pi} = \{A_0, A_1, \ldots, A_{n-1}\}$, all actions in A_i are invoked in parallel by generating an equal number of concurrent *futures*, i.e., containers which act as proxies for the yet unknown result of the respective action. The futures complete either when a response is received (indicating success or failure) or when some predefined timeout period expires. The timeout reflects a short delay within which an average, reasonably fast service is expected to respond. Services which justifiably need longer time for searching or processing data, are kept in a list of pending actors until they respond successfully or they expire, i.e., the respective expected long response time passes. In case an asynchronous mechanism for receiving notifications about environmental changes is available, the assumption is that this mechanism is reliable and timely, so that notifications regarding the effects of the completed futures are received within a few milliseconds after the short timeout period.

After the short timeout elapses, the collected information and context changes are processed to decide on plan refinement or replanning from scratch. The updated plan (refined or new) is computed starting from an initial state that reflects the new context, and *assumes* the successful completion of any pending actions. Thus, considering the invocation of the parallel actions A_i at state i, and some actions $A_p \subseteq A_i$ which do not respond within the short time limit, the new initial state is formed by assigning all variables that participate in the effects of A_p to the values they have at the solver state $i+1$. The values of these variables at state i are stored for bookkeeping so that they can be later recovered in case of a failure response or timeout. An action a which is part of the updated plan is executed only if *precond*(a) does not include any variable which is part of the effects *pendEffects* of any of the pending actions. Moreover, if (i) the goal comprises a goal of type $g1$ `under_condition` $g2$, (ii) some variable in *precond*(a) or *effects*(a) is involved in any of the propositions within $g1$, and (iii) some variable which is part of *pendEffects* appears in $g2$, then a is prevented from being executed. In such a case, the orchestration process waits until the respective pending action either responds or expires, i.e., the expected delay time elapses. Expiration is interpreted as an indication of erroneous behavior and can be dealt with as described in Section 5.4.

5.4. Dealing with erratic behaviors, constraint violations, and persistent information

Erroneous responses and expirations are handled depending on the type of the faulty service, the availability of alternative service implementations, and on the severity of reported failure. Accordingly, a second invocation attempt may be made, an alternative service provider may be looked for, or another plan which can lead to the same results may be computed. A planning action corresponds to an "abstract" service or service *type*, which can be translated to different concrete service operations at execution time. We assume this matchmaking process to be undertaken by a special-purpose discovery and selection component, e.g., [109,101], which returns the set of functionally equivalent services and selects the next appropriate instance to invoke according to some criteria.

In cases where a sensed output leads to a constraint violation, one may seek another concrete instance, retry with the same provider or make a functionally different choice, depending on the nature of the service. For services whose output may differ depending on the selected provider, such as stores returning the availability or price of a requested item, it makes sense to try alternative instances. This is not the case for services that provide information that is not instance-specific, such as the weather, map etc.

If the received faulty response indicates a permanent failure, then there is no use in trying to invoke the same service instance again. The respective service implementation entry in the list of candidate services is marked accordingly, so that the specific instance is restrained from future selections. If no alternative implementation for the same action can be found, a constraint which forbids the action in question to be chosen by subsequent plans is added to the CSP. Depending on the policy and the type of the service, the ban may concern the action in general, i.e., for all $0 \leq i < k$, $a[i] = 0$, or the action in combination with the input parameters values $\overline{v_p}$ it was invoked, i.e., for all $0 \leq i < k$, $\bigwedge_{p \in in(a)} p[i] = v_p \Rightarrow a[i] = 0$. The underlying assumption is that the type of failure and reliability of a service can be inferred by the service's response (error code) in combination with a history of the service's behavior. An external diagnostic system [30], can be consulted for this purpose.

Byzantine services which indicate successful completion without delivering the expected results can be indirectly tolerated without threatening the consistency of the plan, if the orchestrator is consistently and timely informed about the actual state of the world. Consistency checks are performed on the basis of the latest change events, and thus, at each step. The preconditions of the next actions are checked towards the actual instead of the expected environmental state. In this way, for example, before trying to move through a door, the orchestrator will wait until receiving the change event that the door has been opened. Spotting *which* service is the abnormal one is a much more difficult task. Separate dedicated monitoring techniques are required to decide whether a service is byzantine or not, by inspecting the behavior of all services and infer unusual patterns, as e.g., in [91]. To prevent the orchestrator from repeating invocations to malicious services, an upper limit is set to the number of times that a certain operation can be consecutively called with the same input for the same planning state. This is a way of enforcing "fairness" [27], so as to eventually exit the execution loop.

5.5. Implementation

The framework for interleaving planning, monitoring and execution has been implemented by using the *akka* library in Scala (akka.io), which builds upon the theory of actor models [49,54]. Akka provides the means for building scalable and fault-tolerant concurrent applications at a high abstraction level, based on an asynchronous, non-blocking and lightweight event-driven programming model. The orchestration component, the RuGPlanner and the service environment correspond to different actors, which are independent entities that operate concurrently and communicate with each other by asynchronously exchanging messages. Each component-actor may supervise a set of smaller actors-children, each of which represents a lower-level constituent entity and is responsible for certain functions that are assigned to it.

All services that participate in the service environment are modeled as individual actors, which are overseen by the parent environment actor. The parent actor delegates the requests for service calls it receives from the orchestration component to the respective child-actor, which simulates the requested service. The service-level actor processes the message and reacts accordingly, depending on the behavior that we wish to simulate, e.g., by replying with a message that includes output values or indicates a failure, by raising an exception, taking too long to respond or sending no message at all. Each child is treated separately, and several fault handling directives (e.g., stop, resume, restart, escalate) can be implemented by the parent actor, depending on the type of failure and the failing service actor. Service-level actors can be connected to real services, e.g., an OSGi bundle interfacing a physical domotic device [68], or the API of a service available on the Web [120]. Concurrency is dealt with through *futures*, which are used to retrieve the results of parallel invocations in a non-blocking way, as described in Algorithm 1. Each future's lifecycle is treated by a separate callback, which amounts to a generated special-purpose actor that waits and reacts to the future's completion. These callbacks, which may be executed in any order or in parallel depending on the service behavior, entail specific modifications to the CSP, which is shared among them. Atomicity on the operations on the CSP is ensured through the Scala Software Transactional Memory library (nbronson.github.com/scala-stm), which takes care of coordinating access to shared data from concurrent threads.

Regarding constraint solving, we use the Choco version 2 constraint programming library [25], which provides a large choice of implemented constraints, a variety of pre-defined but also custom search methods, and supports the dynamic addition and removal of constraints. In the current implementation, the supported types for state variables are enumerations and integers. In Choco, variables with large domains are only represented by their lower and upper bounds, so that

propagation events only concern bound updates. An integer variable with undefined bounds corresponds to an interval from −21 474 836 to 21 474 836. Real variables are also supported by Choco, however, investigating their impact on planning performance is left as future work.

6. Evaluation

To evaluate the time performance of the planning and orchestration framework of several complex goals, planning domains, and execution circumstances, we consider two test cases: (i) a marketplace of services on the Web, where the planning agent communicates with the execution environment in a synchronous manner, and (ii) a setting where exogenous events are present, about which the planning agent is asynchronously notified. We also perform a number of tests regarding the scalability of the system with respect to a number of factors, such as the number of grounded actions, the variables domain size, and the need for sensing. All scenarios presented in the followings were tested on a Core i5 @2.50 GHz computer with 4 GB of RAM, running Ubuntu 12.10.

6.1. WS marketplace

Consider the example about attending a concert presented in Section 3.2. In such a setting, the only source of information about the context changes are the responses of the service invocations, and one can safely assume that no external actor interferes with the plan. The scenario involves many sensing actions and the plan refinement process has to repeatedly take care of the appropriate instantiation of input parameters. For example, when the information about the upcoming band's performance is acquired, the input parameters of all actions which depend on the concert's date and place have to be instantiated to the outputs of the "getNextEvent" action (instead of the random convenient values they were assigned offline).

We presented a run using actual services on the Web and employing an orchestration algorithm that plans from scratch at every step in [67]. The run shows a situation for which the place of the first upcoming concert turns out to be too far. At the initial state all variables referring to the weather, distance etc., are unknown, since they are not included in *knownVars* (their knowledge is predicated on input parameter values as stated in *knowlBase*).

Since retrieving the facts included in the knowledge base does not lead to a solution, the planner adds the following sensing actions to the plan: first the retrieval of the next performance, and then the respective actions for sensing the new distance, weather, hotel list etc. Refinement is performed once more, after the information about the next performance is instantiated. Then the information about the weather, calendar availability, distance and hotel rooms regarding the new whereabouts is collected in parallel. Since the new information does not violate the goal requirements, the existing plan is found to be consistent, and a ticket is booked. However, when proceeding to hotel room reservation, the first hotel provider in the list (the default order is by increasing price) returns a permanent failure. The policy for dealing with a *bookHotel*'s failure response in this case is to forbid it to be called again with the same input arguments. Thus, the refinement process enforces the investigation of the next provider in the list of available hotels.

The sequence of steps taken by the orchestrator are summarized Example 7.

Example 7
getEventsList(Neutral Milk Hotel) $\rightsquigarrow$ *evList* = known
getNextEvent $\rightsquigarrow$ eventDate = 2014-02-05, *eventPlace* = Brussels
*R*efine plan (assignment of outputs to inputs of actions in plan suffix)
In parallel: {
checkCalendarAvail(2014-02-05) $\rightsquigarrow$ *calendarAvail* = true
getTemperature(Brussels, 05 Feb 2014) $\rightsquigarrow$ *temperature* = 11
getDistance(Groningen, Brussels) $\rightsquigarrow$ *distance* = 360
getAvailHotels(2014-02-05, defaultPl, 1, single) $\rightsquigarrow$ *hList* = known
}
Sensed value 360 for distance violates constraints, *Refine plan*
Updated plan found by adding extra actions to plan suffix
getNextEvent $\rightsquigarrow$ *eventDate* = 2014-02-08, *eventPlace* = Amsterdam
Refine plan (information regarding date and place has changed)
In parallel: {
getDistance(Groningen, Amsterdam) $\rightsquigarrow$ *distance* = 182
checkCalendarAvail(2014-02-08) $\rightsquigarrow$ *calendarAvail* = true
getTemperature(Groningen, 08 Feb 2014) $\rightsquigarrow$ *temperature* = 11
getAvailHotels(2014-02-08, Amsterdam, 1, single) $\rightsquigarrow$ *hList* = known
}
bookConcertTicket(Neutral Milk fHotel, 2014-02-08) $\rightsquigarrow$ **ok**

getNextHotelInfo $\rightsquigarrow$ *hotelWS* = Chancellor Hotel, *hotelPrice* = 60
Refine plan (assignment of outputs to input parameters of actions in plan suffix)
bookHotel(Chancellor Hotel, 2014-02-08, 1, single) $\rightsquigarrow$ **null**
A failure occurred, *Refine plan* (after banning bookHotel with same input parameters)
getNextHotelInfo $\rightsquigarrow$ *hotelWS* = Fairmont Hotel, *hotelPrice* = 75
Refine plan (assignment of outputs to input parameters of actions in plan suffix)
bookHotel(Fairmont Hotel, 2014-02-08, 1, single) $\rightsquigarrow$ *hBooked* = true

The plan is refined whenever the response of an action invocation includes newly sensed output, and the consistency check of the existing plan suffix fails. This happens either because the input parameters of subsequent actions have to be updated, or because the new information violates a constraint. If the response indicates success (**ok**) and the action does not include any knowledge-providing effects, then the next action(s) are executed. A response which indicates a failure (**null**) also calls for plan refinement.

The travel and shopping scenario is a widely used example [110,78,102,67], which we adapt by using real and virtual services, derived from a variety of application domains: making online appointments, shopping, shipping, traveling, learning about entertainment events, and obtaining general purpose information. In such test cases, there are not many structurally different plan-compositions that can achieve a given goal. E.g., when planning for an item's purchase and shipment, the main steps of the composition, like selecting a seller, ordering, paying, are more or less the same for all providers that offer the respective services. If an inconsistency occurs during execution, this can be usually resolved by some extra actions invocations, such as looking for another shop or shipping service. Therefore, time performance is better than a planning-from-scratch approach, e.g., as in [67]. Moreover, the support of parallel execution also contributes to reducing overall orchestration time.

In total, the domain we used consists of 42 operators (amounting to millions of grounded operators), 33 of which are knowledge-providing. The results of running a number of diverse scenarios (combination of goals and execution behaviors) are summarized in Table 1 (the goal "goToConcert" corresponds to the motivation scenario described above). In order to avoid irregularities due to actual response times, all invocations to WSs are simulated. Service responses (sensed information and erroneous behavior) can thus be controlled to test several execution behaviors. All operations are simulated to respond after 1 second, and 1 second is the extra waiting time within which the change events implied by the invocations are expected to be received. The upper time limit for refinement (i.e., before resorting to replanning from scratch) is set to half the time of the last planning from scratch invocation. The time for bootstrapping and transforming the planning domain and goal into a CSP is 1.4 seconds.

6.2. Scalability tests

We test the scalability of the off-line planner with respect to an increasing number of (ungrounded) actions, and an increasing size of input parameters (i.e., grounded operators) and variables. The experiments run on variations of the domain used in Section 6.1, which is typical of service descriptions in a WS marketplace. The goal is a combination of going to a concert, renting a car, and booking an appointment with the doctor, and the plan that satisfies it consists of 14 actions. In Fig. 2, the CSP and plan creation time vs. the number of actions in the domain is plotted. For the evaluation, we create copies of the base domain's variables and actions, i.e., actions with the same logic of preconditions and effects, but on different variables. One notices that for 510 operators and 690 variables, 9 seconds are required to construct the CSP and 96 seconds to compute the plan. It should be emphasized that the size of this domain amounts to an order of 10^{12} grounded actions.

To test the effect of the number of grounded operators on the planning time, we keep the number of operators constant (and equal to 42, as in Section 6.1), and increase the domain size of input arguments and other variables of type integer. The experiments show that the domain size of input arguments as well as of other variables, including knowledge variables, has no impact on planning time. The planning time for a domain with 37 variables having the maximum domain range advised by the underlying constraint solver ($[-21\,474\,836, 21\,474\,836]$) is 3.8 seconds (the same as for variables with domains of

Table 1
Results for the WS marketplace domain (time in seconds).

Goal	♯ actions in initial plan	Refine time (♯ attempts)	Plan from scratch time (♯ attempts)	Consistency check time (♯ attempts)	Total orchestration time
Appointment	5	3.0 (5)	1.2 (1)	5.6 (10)	25
Select&buyCd	6	4.5 (3)	3.2 (2)	6.2 (11)	29
goToConcert	9	6.4 (6)	2.2 (1)	3.5 (9)	23
buyBook&ship	9	10.4 (6)	3.9 (2)	7.4 (10)	38
Travel	15	7.3 (5)	2.1 (1)	4.5 (8)	32
combinedGoal	17	11.5 (10)	2.8 (2)	11.6 (13)	46
		13.7 (15)	2.5 (2)	16.3 (19)	62
		18.2 (19)	4.4 (4)	21.6 (24)	79

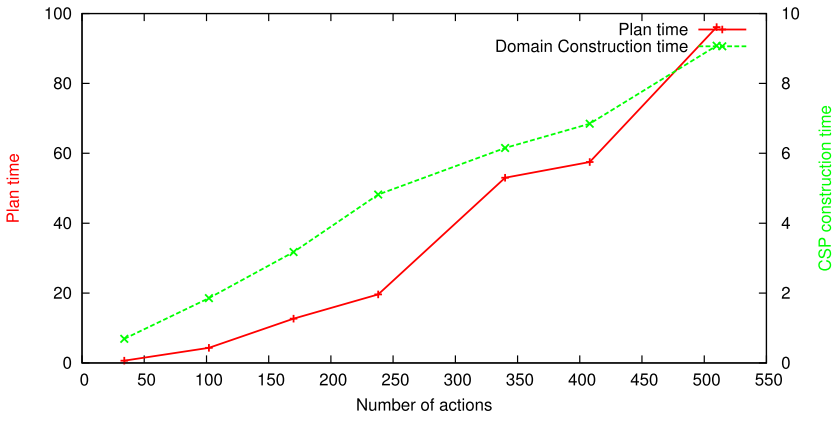

Fig. 2. Time for constructing the CSP domain and time for computing the plan in seconds vs. the number of actions in the domain.

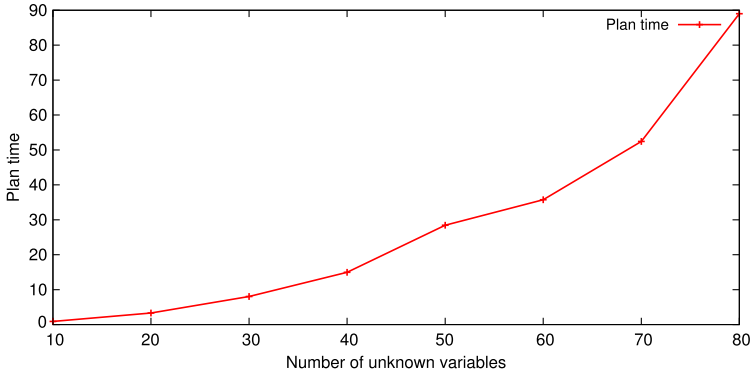

Fig. 3. Time for plan computation in seconds vs. the number of unknowns, where sensing actions are performed in succession.

range 10). Considering that an action in the WS marketplace domain has up to three input parameters of type integer, the number of grounded actions in this case is in the order of 10^{343}. Dealing with such a scenario would be infeasible for a planner resorting to a compilation into a fully grounded encoding.

To further explore the impact of variable domain size, we perform tests for increasing plan lengths. For the evaluation, we construct a domain consisting of an increasing number of interdependent actions, which have to be performed in sequence to achieve the goal. The experiments show that up to a limit of 50 in the number of variables and actions in plan, the planning time is not affected by variables domain range. In a domain consisting of 80 variables and respective actions which affect them, the planning time is 54 seconds for a domain range of $[1, 10]$ and 125.5 for a domain range of $[1, 10^4]$. This result confirms that the most constrained/increasing domain selection strategy is but little influenced by the fact that the number of possible assignments to the CSP is exponential with the variable domain size. An interesting line of future research is the investigation of the planner's performance in case of domains with real variables.

To investigate the impact of the amount of unknown variables in the initial state, we have designed a domain where each knowledge variable is being observed by a distinct sensing action. Fig. 3 shows plan computation time with respect to an increasing number of variables with domain range $[1, 100]$, which all have to be sensed to fulfill the goal. Sensing actions are dependent on each other, so that they have to be performed in sequence in the plan (i.e., plan length is equal to the number of unknowns). Because the notion of state for the RuGPlanner encloses a set of fully instantiated states, the number of possible initial (instantiated) states of knowledge variables and their domain size affects planning time only with respect to the number of sensing actions which have to be included in the plan, their interdependencies, and entailed constraints. This is important, considering that the compilation of the contingent planning problem to a problem which can be solved by a classical planner [2] is, in the general case, linear in the number of possible initial states, and exponential in the number of uncertain fluents. On the other hand, approaches which rely on such translations take advantage of the high efficiency of classical planners.

In order to test the planner's behavior under continual calls for plan revision, we have designed a situation where actions fail consecutively. The domain instance consists of copies of actions with the same preconditions and effects, involving the same variables. At each step of execution, the action responds with a failure, and either plan refinement or replanning is consecutively required. In the first case, each failure can be addressed by adding a different equivalent action copy at the beginning of the plan, while, in the second case, the new plan consists of different action copies. In both cases, the number

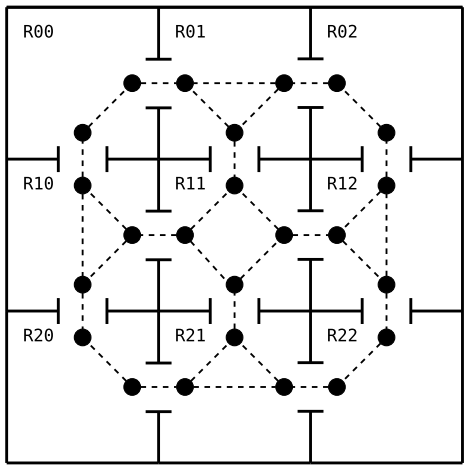

Fig. 4. A 3×3 grid representing rooms and doors which can be navigated by a robot.

of failures does not affect the time required for plan repair or for replanning. For a situation of 50 consecutive failures (and equivalent number of action copies), the time for refining the plan is between 0.8 and 1.1 seconds. Similarly, the time for computing each new plan from scratch is the same as the time for computing the original plan.

To sum up, we have tested the performance of the RuGPlanner with respect to different factors. The evaluation demonstrates that RuGPlanner can efficiently deal with very high cardinalities (in the order of 10^7) of input parameters to operators for typical data-intensive WS domains. The benefits of non-grounding become clear when considering that the same domain would translate into a prohibitive number of grounded action instances. Moreover, working at the level of set of states enables RuGPlanner to accommodate for a large amount of uncertainty, i.e., the number of fully instantiated states that correspond to the uncertain state and the number of unknown variables.

6.3. Robot moving in a grid

To further investigate the behavior of the RuGPlanner in an environment which calls for continuous plan revisions, we use an example that combines in a single run of the orchestrator three types of run-time contingencies: an exogenous action which interferes with the plan, a malicious service with byzantine behavior, and an operation with a long response time. The example concerns a robot moving around in a house setting as depicted in Fig. 4. The setting is an IPC-like domain, in which the RuGPlanner cannot compete with approaches which translate the contingent version of the problem into a classical planning problem, and, therefore, scale more efficiently with the size of the grid. However, the experiments on such a domain demonstrate the generality of RuGPlanner which can deal with all kinds of contingencies which previous approaches ignore, and help us explore the tradeoff between plan refinement and replanning from scratch.

The rooms in the grid domain are connected through doors which can be open, closed or locked. The robot can open a door only if it stands in front of it on either of the two sides, and only if the door is unlocked. For opening the two doors leading to room $R22$, the robot needs to have at hand a special password to be retrieve by invoking a slow sensing operation that requires 40 seconds to respond. The robot can move as instructed by the action presented in Example 1, that is, between locations which are adjacent to each other (connected through the dashed lines in Fig. 4), with the additional requirement that if the locations belong to separate rooms, the intermediate door should be open.

Initially, the robot resides at location $R00\_R01$, and the goal is to reach location $R22\_R21$ at the final state (where *RX_RY* is the location at room *RX* and is connected with a location in room *RY*). All doors are initially closed and unlocked. The initial plan guides the robot to $R22$ through $R01$, $R11$ and $R21$, and instructs calling the sensing action for retrieving the necessary password at the second state of the plan. Since the sensing action takes long to respond, it is checked whether there are any actions in the plan suffix that do not depend on the password retrieval. Indeed, the robot can keep on moving till the door leading to $R22$. However, while the robot is still in $R01$, someone locks the door leading from $R11$ to $R21$. Such a contingency requires a drastic change in the robot's planned route, and cannot be dealt with by just augmenting the plan. The refinement process fails, and replanning from scratch is performed.

The new plan leads the robot through $R02$ and $R12$. Due to bad luck though, it turns out that the door that connects $R02$ and $R12$ behaves unreliably: although the opening operation responds with a success, the door is actually not opened. As a result, an augmented plan which includes a second attempt is computed. The door behaves the same way for a second time, and thus a new refined plan is produced. However, the opening door operation is not invoked again, since already two invocations corresponding to the same planning state have been attempted, and the action is in turn forbidden from any subsequent plans. Since no refined plan can be found given the prohibition of the specific door opening operation, a new plan is computed from scratch. The alternative route directs the robot back to $R01$, then to $R11$ and then to $R12$. However,

Table 2
Test results for the robot in a room scenario of varying grid size (time in seconds). Total orchestration time counts the time elapsed between issuing the goal and its satisfaction or failure, including time required by actions to execute and respond.

Test	Refine time (♯ attempts)	Plan from scratch time (♯ attempts)	Consistency check time (♯ attempts)	Total orchestration time
unlock3	0.9 (3)	0.3 (1)	0.4 (11)	24
unlock3*	–	1.0 (4)	0.5 (13)	25
unlock4	2.9 (5)	3.3 (1)	3.5 (26)	62
unlock4*	–	12.8 (6)	3.2 (24)	69
unlock5	6.0 (7)	39.2 (1)	10.3 (24)	102
unlock5*	–	202.6 (10)	12.4 (33)	327
unlock-notInRoom3	0.8 (3)	0.6 (1)	1.2 (18)	39
unlock-notInRoom3*	–	2.4 (4)	1.3 (18)	41
unlock-notInRoom4	3.9 (5)	3.2 (1)	4.1 (21)	53
unlock-notInRoom4*	–	19.5 (6)	5.2 (26)	79
unlock-notInRoom5	13. (7)	59.3 (1)	22.8 (33)	158
unlock-notInRoom5*	–	176.8 (8)	23.0 (33)	268
permanent-lock3	0.2 (3)	0.3 (4)	0.3 (8)	18
permanent-lock3*	–	0.4 (4)	0.3 (8)	18
permanent-lock4	13.4 (8)	8.1 (9)	5.2 (23)	61
permanent-lock4*	–	8.3 (9)	4.9 (22)	49
permanent-lock5	81.5 (12)	147.1 (13)	13.8 (27)	317
permanent-lock5*	–	135.6 (13)	14.3 (31)	243

as the robot stands before the door leading to *R*22, the password sensing action expires (it has not responded within the expected maximum timeout), and is thus treated as a failed action. Since there is no other way to collect the password, it is impossible to satisfy the goal. An alternative policy could be to try re-invoking the expired actions, with the hope that they would provide the required output.

During the above orchestration run, there are 14 validation checks and 4 attempts to refine the plan (including the failed ones), taking 6.6 seconds in total, while 3 plans are computed from scratch (including the initial one), taking 18 seconds altogether. One notes that the evolution of the orchestration run actually highly depends on the structure of the generated plans. For example, in the above example, assuming that doors can be opened remotely, independently of the location of the robot, it makes a difference whether the opening of the doors is scheduled in parallel at the beginning of the plan, or delayed till later steps. Although both plans consist of the same number of actions (i.e., are equally "good"), it is desirable to push actions as early as possible in the plan, since in this way any erroneous service execution can be detected earlier. Moreover, operations which are expected to take a long time to complete should be preferably scheduled at an early stage of the plan.

6.3.1. Refinement towards replanning from scratch

The test cases presented in Table 2 investigate the tradeoff between attempting a plan refinement versus planning from scratch. All tests are variations of a planning domain which represents a robot moving between adjacent rooms connected by doors, which can be open, closed or locked. In all tests, the goal is for the robot to move from the top leftmost room to the bottom rightmost room, starting from an initial state where all doors are closed and unlocked. Execution times are expressed in seconds and the total orchestration time counts the time elapsed between issuing the goal and its satisfaction or failure. Plan from scratch time includes the time for computing the initial plan. The numbers 3–5 indicate the dimension of the grid, i.e., refer to a 3×3, 4×4 and a 5×5 grid, respectively. Tests marked by "*", no attempt for plan refinement is made, and all updated plans at every step of the orchestration process are computed by resorting directly to replanning from scratch. The upper time limit that the refinement process may take is set to half the time required by the last planning from scratch invocation.

In all runs, as the robot proceeds, an external actor repeatedly hinders its route: just before the robot is about to cross an open door, the troublesome agent locks that door. All operations are simulated to respond within 4 seconds, with 2 seconds of extra waiting time, within which the change events implied by the invocations are expected to be received. The conservative most-constrained/increasing domain strategy is used for planning from scratch in the 3×3 and 4×4 grid tests, and for the refinement process in all cases. For the tests in the 5×5 grid, the initial plan takes more than 15 minutes to complete with the default search strategy, and therefore a random value assignment approach is employed whenever planning from scratch, with the search restarting every time a fixed increasing node limit is reached. The resulting plans are not always the optimal ones, and may include redundant actions.

In the tests signified by "unlock", the robot can open as well as unlock rooms. Thus, every time the robot unexpectedly encounters a locked door, it has to update its plan by first unlocking and then opening the door targeted by the exogenous agent. In these cases, attempting a plan refinement instead of directly planning from scratch saves a considerable amount of time, at least for the 4×4 and 5×5 grids. As expected, the longer time plan generation takes, the more benefit is gained by employing plan extension, since in these simulations the updated plan should just involve the addition of two more

Table 3
Results for tests with long sensing actions (time in sec).

Test	Refine time (♯ attempts)	Consistency check time (♯ attempts)	Total orchestration time
pswdOpen3-opt	0.5 (1)	0.6 (7)	76
pswdOpen3-subOpt	0.8 (2)	0.9 (9)	88
pswdOpen3-worst	4.7 (4)	1.1 (11)	121
pswdOpen4-opt	4.1 (1)	10.8 (8)	113
pswdOpen4-subOpt	11.0 (3)	18.3 (13)	155
pswdOpen4-worst	12.1 (6)	13.7 (22)	275
pswdOpen5-opt	12.8 (1)	58.5 (11)	144
pswdOpen5-subOpt	33.5 (3)	60.9 (14)	281
pswdOpen5-worst	63.4 (8)	71.6 (20)	437

actions at the beginning, which is very fast to compute. The two approaches also lead to slightly different overall sequences of steps, since some redundant actions may be included at times, especially when employing the random value assignment strategy.

To investigate the performance of the orchestration in situations where a more elaborate plan refinement is required, let us assume that the robot can unlock a door only if it resides in a room different from the ones the door connects. Whenever the exogenous agent locks a door that the robot is about to cross, the robot has to first move to some adjacent room, unlock the door, then move back, open it and go on with its route. This situation is reflected by the "unlock-notInRoom" tests in Table 2. Also in these tests resorting to plan refinement leads to considerably better performance. However, in cases where there is no way to augment the plan to tackle the new contextual situation, the time devoted to attempting plan refinement is wasted. This case is simulated by the "permanent-lock" tests, which concern the same planning domain as in "unlock", except that the robot has no capability to unlock doors. Thus, every time the robot has to deal with an unexpectedly locked door, the refinement attempt fails, since a drastically different plan has to be found. In this case, resorting directly to replanning from scratch actually saves time.

The experiments confirm that the balanced approach between attempting plan refinement and planning from scratch yields best results when minor changes to the already existing plan are enough to resolve the inconsistencies due to incomplete knowledge or external events. In cases where radical plan revision is required, skipping over the plan refinement phase yields better time performance. The RuGPlanner adopts a middle-ground approach based on the following rule of thumb: if the plan revision process takes too long, this is probably an indication that the new situation calls for a plan that looks quite different from the original one, and should therefore give up in favor of replanning from scratch.

6.3.2. Timeout of sensing actions

The next tests simulate situations where some actions take justifiably long time to respond, and aim at demonstrating how different plan structures affect the overall orchestration process. In these tests, to open a door the robot should know a 3-digits password specific for that door, which it can retrieve by invoking a sensing action from any location. The robot has to move from the uppermost left room to the bottom rightmost one, starting from an initial state where all doors are closed and all passwords are unknown to the robot. In all tests simulating the execution behavior, the password sensing actions take 40 seconds to respond, moving actions take 8 seconds, opening the door takes 1 second, and the average waiting time for a bundle of parallel actions to execute is 10 seconds.

Depending on which planning states sensing actions are scheduled, the robot may end up waiting for a shorter or longer time for a sensing action to respond. Table 3 summarizes the results for three different structures of initial plans passed for execution and monitoring to the orchestration algorithm. All plans consist of the same number of actions, however the state at which actions are placed varies: "opt" refers to the optimal situation where all the password sensing actions in the plan are concentrated at the first state, "subOpt" to the plan actually generated by the RuGPlanner by employing the random values assignment with restarts searching strategy, and "worst" to a plan where each password sensing action is scheduled just before the respective door opening. After the invocation of some parallel actions sets, a validation check and possibly a refinement attempt is performed when all actions respond or the short timeout expires. Given the validated or updated plan, it is checked whether it is possible to proceed with any plan actions which do not require the knowledge of the specific password. Thus, the robot can move further if possible, while the password is being sensed (this happens with the "subOpt" simulations). Whenever a pending action responds, the orchestration goes on with executing the updated plan suffix. The results in Table 3 demonstrate the large gain in time when the actions which take long to respond are scheduled as early as possible in the plan. The larger the domain, i.e., the more slow actions are involved, the larger the difference between the optimal and the worst approach is.

6.3.3. Discussion

The evaluation demonstrates that the RuGPlanner performs well in domains which are too large to ground, and which involve variables, and especially observation variables, with large domains.

Producing the shortest plan is not enough to lead to the optimal orchestration runs. The plans that are opted for should be the ones which include actions as much in parallel and as early as possible. However, the RuGPlanner does not always produce the desired structure of plans, and in some cases it even computes suboptimal ones. Moreover, the orchestrator may be trapped into repeating a sequence of steps without managing to reach the goal. This "stuck in a loop" situation may arise in cases where suboptimal plans of a certain pattern are repeatedly produced, or due to the malicious behavior of external actions which may lead a certain execution into a deadlock. To give an example, assume a robot that wants to move from $R00$ to $R02$ of Fig. 1, and that the door leading from $R01$ to $R02$ can be opened only if the robot resides in $R00$. If an external actor closes the door leading from $R01$ to $R02$ every time before the robot tries to pass through it, a revised plan instructing the robot to go back to $R00$, reopen the door and move forth again will be repeatedly generated. The orchestrator has no way to identify that it is trapped in a loop due to the consistent behavior of an exogenous actor, so as to make the decision to follow an alternative plan, e.g., reach $R02$ through rooms $R10$, $R11$, and $R12$, with the hope that the doors in that route will not be blocked. A similar deadlock may result in any case in which there is need for replanning when the robot resides in $R01$ and the planner generates a suboptimal plan, which directs the robot first back to $R00$, and then forward to $R01$ and further. To avoid such situations, some randomness should be inserted in cases where the orchestrator repeats the same sequence of actions, and this repetition is not included in the current plan. For example, the orchestrator may try to randomly choose from the set of actions applicable at the current state, with the hope that from that new state the planner will escape the deadlock situation.

7. Related work

A great number of approaches have been proposed in the literature about describing, constructing, executing and maintaining Web Service compositions, with research approaching the topic from different viewpoints, including issues related to service discovery and matchmaking, e.g., [109,101], Quality of Service requirements, e.g., [5,51,124], and support for dynamic reconfiguration, e.g., using the channel-based coordination language Reo [81,75]. Several methodologies inspired from work in AI planing have been applied to deal with WSCs. However, because there is no commonly agreed definition of service composition as a planning problem, it is difficult to compare quantitatively approaches proposed in the literature. Some proposals use domain-independent planners [107,88], however their applicability is limited to very simple service domains. The PKS [100] (Planning with Knowledge and Sensing) planning system is used for generating compositions at the knowledge level [85]. Services are modeled as primitive actions specified in a STRIPS-like formalism, however domain-specific design rules are required to capture additional effects triggered by sensing actions and search control constraints. A Partial Order Planning approach is adopted in [98], while in [60] an adaptation of the FF (Fast Forward) planner [59] is used to construct Business Processes from atomic IT entities described in a planning-like manner.

Another line of research builds on modeling the WS domain in the Situation Calculus [89,90,110,111]. The idea is to describe a set of user objectives in terms of a sufficiently generic Golog program, and, the task of composition amounts to the customization of this generic template at runtime with respect to specific user constraints and preferences. Hierarchical Task Networks (HTN) [37] have also been used as a means to represent generic procedures, e.g., [121,108,76,4,83]. SHOP2, a highly optimized HTN planning system, is used to decompose process models into primitive operators/atomic services. SHOP2 does support numeric variables, however, no tests regarding scalability in terms of domain cardinality, especially with regard to unknowns, are provided. In [76], information gathering is performed at planning time, by issuing a list of appropriate queries to collect all the information that is missing in the initial state. Non-blocking strategies for performing search while waiting for queries to respond, similar in principle to the approach employed by the RuGPlanner, are utilized to improve the time for computing a plan. An extension of the algorithm for dealing with information that may change during the operation of the composition is presented in [4], by considering that a solution is correct only within an expiration time. A hybrid approach which combines domain-independent planning with HTN is adopted in [71,72].

As opposed to approaches that view services as atomic planning operators, there is a research track which considers stateful services, where behavioral descriptions impose constraints on the possible interactions that a service can be engaged with. In [113,104,102,103,18,19] component services are seen as state transition systems, e.g., derived from a BPEL description as explained in [113,104], where transitions correspond to asynchronous message exchanges resulting from an atomic action execution. The requirements of the desired composite service are expressed in a temporal logic-like language, and symbolic techniques inspired by model checking are used for computing an executable process. In [17], the approach is extended to support requirements about how to handle uncontrollable events, such as a flight delay.

Another interesting approach which abstracts services as transition systems is the so-called "roman model" presented in [13,16,14,15,28]. From that perspective, the composition problem is treated as a problem of coordinating the executions of a given set of available services described as finite state automata. The objective itself is described in terms of a target service-transition diagram that conforms to some desired interactions.

7.1. Planning domains and goal specifications

PDDL (Planning Domain Description Language) has become the standard language for defining planning problems. It is used in the International Planning Competitions since 1998 [87], and has undergone several extensions [36,39]. PDDL and its extensions represent the world using objects and predicates, and numeric expressions are constructed by using functions

which associate objects with numeric values. Preconditions on numeric fluents are comparisons, and effects can assign, increase, decrease or scale-up/down the values of numeric functions, however, numeric expressions are not allowed to appear as arguments to predicates or values of action parameters. In contrast, the RuG planner is based on a variable/value domain representation which is similar in concept with the Multi-valued Planning Task (MPT) encoding [53,52]. An algorithm for automatically translating a PDDL domain description into a MPT one is described in [53].

The extended declarative goal language supported by the RuGPlanneras described in Section 2.3 enhances the traditional specification of a goal as a set of final states by providing a number of additional features that allow the expression of constraints over state trajectories and hands-off observational requirements. A short overview of its basic operators has been presented in [66]. Many elements of the language are inspired by XSRL (XML Service Request Language) [1,78] for formulating complex requests against standard business processes.

PDDL3 [38] extends PDDL, by supporting a richer goal language which provides for state trajectory constraints which should be respected by the entire sequence of plan states, as well as with soft goals which are desired but not necessary to achieve. The goal language supported by the RuG planner is less expressive than PDDL3, and does not capture preference goals (although the `under_condition_or_not` goal operator could be seen as a form of soft requirement), but several parallels can be drawn with some of PDDL3 modal operators, such as always, sometime, and sometime-before.

The RuGPlannergoal language shares many concerns with the work presented in [48,44,46], which deals with meeting user goals in environments similar to the Unix operating system. Since incomplete information is intrinsic in such domains, distinguishing between satisfaction and mere observational goals is essential. A clear distinction between achievement and information gathering goals is also kept in [98], for the purpose of composing semantically annotated WSs transformed into PDDL operators.

Systems that follow the planning as Model Checking approach have built-in support for temporally extended goals, which allow imposing constraints on the state trajectory, e.g., specification of safety or liveness properties. In [113,104], the EAGLE goal language, based on temporal logics extended with preferences, is used for composing WSs modeled as state transition systems. Several goal specifications for composing WSs move away from the purely domain-independent declarative spirit, and require that the set of possible solutions is pre-defined in some form of procedural template, either in the form of HTN methods, e.g., [4], as a Golog program, e.g., [110], or as a target state automaton, e.g., [12]. In such a context, runtime synthesis is responsible for customizing the high-level procedural specification with respect to user constraints and preferences. From that perspective, the work in [110,111] extends the approach presented in [110], so that Golog generic procedures can be customized not only based on hard but also on soft constraints, yielding compositions which are optimal with respect to the latter. An interesting proposal for fusing procedural and declarative goals to allow greater flexibility in expressing goals is made in [106]. The goal language supported by the RuGPlanner, on the other hand, only specifies *what* has to be achieved but not *how*, i.e., no search control knowledge is used to guide the planning process, neither in the form of control rules or in the form of task decomposition.

7.2. Planning with incomplete information and sensing

Service environments are inherently uncertain about the initial state and unpredictable in their run-time behavior. In the XLPLAN planning system [71,72], external procedure calls are implemented through linked call-back functions, which return a Boolean indication of whether a predicate has been added or deleted from the next world state. In [60] non-determinism stemming from the set of alternative action outcomes is treated through “determinization”, i.e., each non-deterministic action is compiled into a set of deterministic actions, one for each possible outcome. Although performance may be acceptable for binary variables, strategies that resort to determinization are not effective when the cardinality of possible outcomes increases.

Some approaches address the problem of incomplete information by only simulating world-altering effects during the composition process, assuming complete independence between sensing and world-altering actions and setting limitations on the interleaving between knowledge-providing and world-altering actions. In [90,110], information providing services are modeled as external function calls within the Golog programs. The approach relies on the assumption that information persists for a reasonable amount of time (until all actions that make use of it are executed), and that it is not altered by any subsequent action inside or outside the composition. It is also taken for granted that all sensing actions can be performed even if the world-altering effects of actions that precede them in the plan have not been materialized (but only simulated). Similarly, in [76,4], information gathering and execution is treated as a task disconnected from planning, and execution is ceased until all sensing actions return. Analogous assumptions are made in [97,98], where the subset of the plan consisting exclusively of sensing actions is extracted and executed first.

Algorithms for searching at the knowledge level have been proposed by the research line focusing on composing services as state transition diagrams, based on the use of Binary Decision Diagrams. One of the shortcomings of the initial algorithm presented in [104] is that it can only deal with Boolean-valued data. Extensions for non-Boolean data are considered in [102,18], which however suffer from degrading performance as cardinality grows. As shown in [19], belief-level construction grows exponentially with the branching factor of the conditional solution. Given the large number of possible outcomes of sensing actions and unforeseen contingencies, planning for all potential circumstances that may appear during execution is not a recommended strategy for most service domains.

A different knowledge-level formulation as instructed by PKS is used in [85]. Although the version of PKS used in [85] cannot deal with high ranges of possible outcomes, it would be interesting to investigate the performance of the extended PKS presented in [99], which allows the generation of conditional plans that cover numeric-valued outcomes by means of interval-valued functions.

Since almost all state-of-the-art planners resort to some kind pre-processing for compiling the PDDL domain into a fully grounded encoding, on-the-fly handling of runtime outputs is difficult to implement. The problem of incorporating data production and flow into a plan has been investigated in [45,47]. Although [47] considers a planning graph approach, its basic idea of adopting a CSP encoding which amounts to a lifted (not grounded) representation is also adopted by the RuGPlanner. In [56], data production is addressed by considering sets of additional potential constants to instantiate outputs, and by applying an adapted version of conformant FF [22]. Input-output matchings are dealt with based on axiomatizations [57,55] describing the ramifications entailed by sensing, i.e., implications entailed by the outputs/newly created constants of services/operators. However, the approach is limited to propositional effects, and the problem of search space explosion when considering many output constants remains.

Independently of the problem of WSC, there have been large advances in the performance of contingent planners which operate under uncertainty. For example, subtle logical formulas have also been applied for the compact representation, pruning and search in AND/OR graphs at the belief state space [112]. Instead of an explicit encoding of all possible states, some approaches advocate an implicit representation of beliefs by keeping a history of actions and observations made, and inferring from them whether or not a proposition holds, e.g., [58,105]. Interestingly, an approach for translating both contingent and conformant problems (i.e., the case where no sensing is available) into a classical planning problem over the state space rather than the belief state has been proposed [96,2,3]. However, these contingent planners are not tested in domains where the outcome of sensing actions ranges over large domains and may be passed as input to subsequent actions, while erroneous situations and interference by external actions are not considered. An action language that provides for sensing actions with probabilistically and qualitatively non-deterministic effects is proposed in [63], and belief graphs are used to compute conditional plans. However, to the best of our knowledge, all these versions of contingent planning only consider observational effects that are propositional. If the application domain is characterized by an intractably large set of contingencies, a plan monitoring and repair approach is probably more appropriate.

7.3. Replanning and interactions with the environment

In dynamic and uncertain domains, acting, sensing and planning have to be intertwined, imposing that the plan is continuously adapted to the knowledge acquired during execution. CIRCA [93] is one of the earlies implementations of reactive planning systems, based on a nondeterministic finite automaton derived from a world model representation of safety conditions and possible state transitions. State changes entailed by events are prespecified as part of the world model. To deal with the state explosion problem, a method of abstracting variables and states is used [92], without, however, providing any evaluation or considering domains with numeric aspects. SimPlan [64] models the set of possible execution sequences as a stochastic automaton, with exogenous events being represented as non-deterministic transitions associated with a probability. The method makes use of extra guidelines, expressed in temporal logic formulas, for avoiding unnecessary state expansions. Although no evaluation is provided, the model is expected to suffer from state explosion.

Some planning approaches to WSC provide simple reaction mechanisms to contingencies, which are usually hand-coded and domain dependent. In [98], a partial order planner is used, and success conditions are included in actions' effects specifications. Replanning is triggered whenever a causal link indicating an interdependency between actions is violated due to an inconvenient outcome at runtime, and those violated links are avoided by the replanning search process. In [73], the XLPLAN planning system is extended with an event listener about new facts, and offers some replanning capabilities, relying however on a closed-world assumption. In [17], exogenous events are treated via reaction goals, which state what should be done when certain actions take place. The computed composition is a conditional, tree-structured plan, including branches regarding recoverable goal states. The approach suffers from performance problems when the branching factor grows. Markov Decision Processes (MDP) constitute an established mathematical model for probabilistic planning problems, and there are many planners which deal with probabilistic models, e.g., [20,122,43]. Replanning has been extensively employed by approaches which work on the determinized version of probabilistic domains, like FF-Replan first presented in [59], and further extended, e.g., in [123]. A common principle that is shared between FF-Replan and the RuG planner is that both rely on optimistic assumptions about the future, i.e., they compute a solution by selecting the most convenient outcome. A strategy for identifying actions with unrecoverable outcomes, and adding precautionary actions to the optimistic plan so as to avoid dead-ends is described in [34]. A Monte Carlo technique and an approach of casting the problem to a Partially Observed MDP, for re-evaluating the utility function of the continuous multi-dimensional resource space in face of execution-level contingencies is proposed in [11].

There are several approaches that work on probabilistic domains with partial observability and sensing actions. In [105], the replanning approach for the determinized representation is extended for such domains. During plan execution, belief states are updated accordingly, and replanning is triggered if the initial belief state sampling is not consistent with the world. The approach has not been tested in domains with numeric observation effects. A framework that switches from classical planning to planning in small abstractions of the problem when encountering a sensing action with uncertain outcome is proposed in [43]. This approach can deal with noisy sensors, but the set of outcomes is kept small.

Many approaches to replanning try to reuse parts of the existing plan to guide the search for the new one. The idea is that under certain circumstances the work of adapting the current plan requires less time than planning from scratch, without sacrificing quality. A refinement heuristic for partial order plans is proposed in [115], which involves removing potentially problematic actions from the current plan, and incrementally adding extra actions to it, until reaching a valid plan. An approach that focuses on preserving plan stability, i.e., replanning with minimum changes to previous plans, is presented in [35]. However, depending on circumstances and the kind of changes in the state of the world, the work required for repairing an old solution may be greater than planning by completely disregarding the previous solution [95]. A balanced approach between replanning from scratch and plan repair is proposed in [21], where the plan is used as a bias to the heuristic search for the new problem. In this case, search expands by probabilistically choosing between heuristic search for the new goal and reuse of actions and goals of the past plan. The approach is used to speed up the planning time for classical deterministic domains. All above approaches are propositional.

Previous frameworks that tightly integrate planning, monitoring, execution and information gathering include [48,44,46, 74], which are concerned with building planning agents for dynamic and uncertain environments. The RuGPlanner shares many principles with this work, regarding tractable closed world reasoning with updates, knowledge preconditions, and observation effects that assign values to runtime variables. In [86], a framework which combines a temporal planner, ix-Tex [41], with an execution controller and plan repair and replanning mechanisms is presented. The idea of balancing between plan repair and replanning is in principle similar to ours. The ixTex planner uses CSP techniques to deal with timepoints, numeric variables and execution-level constraint violations, although starting from a different action and change model than ours. The use of a Partial-Order Causal Link approach limits the efficiency of the framework as the search space grows. In the context of WSC, a generic algorithm which performs continual replanning from scratch after every invocation of a knowledge-providing action is described in [79,80].

More recently, a continual planning framework for multi-agent planning under incomplete knowledge has been presented in [23]. Decisions depending on yet unknown facts are postponed through the use of assertions, special virtual actions that trigger replanning whenever their knowledge preconditions are achieved at execution time. Replanning annotations in that context lead into postponing sensing till just before the actions that need the information to be observed, which can be inefficient in terms of total execution time if a lot of time-consuming sensing is required. Interestingly, assertions can also be used to learn new operators that become available to the planning agent during execution. Similarly to the representation adopted by the RuGPlanner, multi-valued state variables are used to model the domain rather than a propositional encoding.

7.4. Planning as CSP

Constraint-based techniques have been used extensively for scheduling problems that reason about time and resources, e.g., [26,10,77]. Applications of constraint satisfaction approaches to planning, like the one we use herein, are less mature [7]. A direct transformation of the planning problem into a CSP has been presented in [42], where constraints describe the preconditions and effects of actions along with frame axioms. Alternatively, in [84] the authors propose a formulation based on successor state constraints similar to the ones captured by the planning graph, yielding improved performance. In [8], enhanced reformulations based on multi-valued state variables and transformations to ad-hoc tabular constraints are applied. Several techniques that aim at reducing search space and improving the efficiency of search strategies have been investigated in [9]. A CSP encoding for producing parallel plans is proposed in [6], through the use of constraints that model the synchronization transitions that are possible between assignments to the same state variable. A compilation of GraphPlan's planning graph into a CSP and the use of constraint satisfaction search techniques to improve Graphplan's backward search has been proposed in [70,29]. Constraints have also been used in the context of partial order planners [116].

Mixed CSPs, which distinguish between controllable decision variables and uncontrollable parameters corresponding to environmental uncertainty and contingent events, have been used for modeling domains with incomplete knowledge and contingent events [32,50]. In [50] mixed CSP is used for solving a control problem for the aerospace domain. Although the planning problem is rather particular and defined in terms of constraint-based automata and environmental constraints, an interesting online solution is followed. New contingent plans are built from scratch incrementally for increasing planning horizons/points in time, and these plans provide decisions for an increasing subset of possible world states. A CSP encoding for the conformant probabilistic planning problem, with no observability and probabilistic actions, is used in [61,62]. An approach that integrates constraint-based reasoning into the planning graph for temporal domains with predictable exogenous events that happen at known times is described in [40].

To the best of our knowledge, CSP-based planners have been used so far for generating offline plans for grounded propositional domains, and are decoupled from the execution environment. The suitability of constraint solving techniques for domain-independent planning in a setting that combines uncertainty, sensing, and unpredictable external events has not yet been investigated. In such a context, dynamic CSP and solution reuse/repair techniques, which use information collected from previous searches to speed up the search in the altered CSP, may prove helpful. This method, which makes use of no-good recording, is proposed in [115]. Performance improves when solving a CSP that differs by one constraint (added or removed) from a previously solved one. In [118,117], heuristics that exploit information about certain important features of the CSP that are not affected by the alterations are used, yielding considerably better performance for randomly changed CSPs.

8. Conclusions

In order to function in service domains, a planning system should be equipped with a number of special features that enable it to dispose of the uncertainty deriving from the open world assumption as well as from unexpected service behaviors, deal with the abundance of data-intensive operations, and overcome inconsistencies due to changes caused by exogenous factors. Most importantly, the applicability of the planning system should not be tailored to the specifics of a particular domain and task, but rather be in the position to fulfill a variety of diverse objectives with minimal manual reconfiguration. To meet these requirements, we proposed and described the RuGPlanner, a planning framework which uses constraint satisfaction techniques and accommodates for complex goals, a knowledge-level representation to model lack of information, proactive sensing in the presence of variables that range over large domains, as well as an algorithm for monitoring execution and revising plans in a seamlessly changing environment.

The approach relies on a domain-independent representation, which consists of a set of loosely-coupled atomic service operations described as planning operators. Data-intensive domains which involve many operations that work with numeric-valued variables, including the case of numeric-valued sensing outputs, are effectively dealt with. Regarding goal expressivity, the RuGPlanner supports a number of constructs that impose constraints over the state traversal, but it does not accommodate for preferences. No restrictive assumptions about the interdependencies between sensing and word-altering actions in the composition are made, and sensing actions are proactively planned for. The approach performs continuous plan revisions to dispose of failure responses, long responses and timeouts, and with exogenous events. The orchestration algorithm can address effectively many problematic situations, under certain assumptions regarding the kind of goal and the point at which the contingency occurs during execution. Experimental evaluation confirms the feasibility of the approach in several situations, with complex goals, and diverse combinations of unknowns, failures and runtime inconsistencies. Although our work is inspired by applications in the field of Web Services, the essence of the planning methodologies we describe is more general, and touches upon issues that concern a series of problems where domain-independence, uncertainty and dynamicity are at stake.

To improve the performance of the RuGPlanner one should consider planning-oriented rather than just CSP-based heuristics, which manage to extract additional constraints that reflect particular properties of the underlying planning problem. Some of the reformulation techniques proposed in [8] could also be used to speed up search. The RuGPlanner generates plans that are usually characterized by a high degree of parallelism, however, special heuristics need to be investigated if plan optimality is at stake. Extending the planning system to deal with noisy data, i.e., sensing actions which return a set of possible values, is also an interesting direction for future work. To this end techniques similar to the interval-valued function described in [99] and some sort of case-based reasoning can be adopted. The capabilities of the orchestration framework can also be improved and extended. For example, exploiting techniques used in the context of dynamic CSP for intelligent solution reuse probably benefit performance.

Appendix A. Orchestration algorithm

The orchestrator is realized as an actor [49,54], i.e., an object which seamlessly reacts in a concurrent manner to messages it receives asynchronously (Algorithm 1). The message newModel(*domain*) is related to the construction of a new CSP model which encodes the planning domain *domain*. The resulting constraints are propagated, and the propagated generic world instance is stored as a starting point for all subsequent solving requests (as long as no service with new functionalities is installed, in which case the model has to be re-constructed). Each time a request for satisfying a new goal is issued (newPlan(*goal*)), the constraints modeling the goal are added to the constraint network (after removing any constraints modeling previously handled goals), and the CSP is propagated and stored as the goal-specific world instance from which search will start in all refinement and replanning attempts, till the goal is satisfied or no solution can be found.

Other messages concern the receipt of a change event, an indication that some service type has become unavailable, and the appearance of a new service type *contextChange*(var, val), *removeAction*(a), and *addAction*(a) respectively. In the latter case, if the description of the respective action a already exists in the planning domain, i.e., the constraints which represent its preconditions and effects are part of the CSP, then it is enough to remove the constraint that had banned it at the solver level. Otherwise, if the new service type offers new functionalities, which have not been encountered before, the addition cannot take place in a dynamic manner: the whole CSP has to be reconstructed, so that the state variables and frame axioms associated with the new action are taken into account.

The message monitor($plan, s_i$) refers to the core part of the orchestrating process. The algorithm represents the most general case, i.e., it considers asynchronous context changes, accommodates for a particular kind of byzantine behavior, and maintains an evolving set of alternative service instances for the same action. The pieces of code which are within curly brackets ({}) indicate critical sections. Since all concurrent futures work on the same CSP and context, only one such entity at a time is allowed to access the CSP or context, and any other critical requests from other actors are suspended until the lock on the respective object is released.

Algorithm 1 Orchestrating actor: asynchronous context changes, concurrent execution and continual refinement of plan or replan depending on latest context.

```
function RECEIVE
  case newModel(domain):
    FORM_CSP(domain)
  case contextChange(var, val):
    {UPDATE_CONTEXT(var, val)}
    if var is part of the effects of some f ∈ pendingActors then
      BOOKKEEPVALUES(f, var, val)
  case removeAction(a):
    Add constraint a[i] = 0 for all 0 ≤ i < k to model level
  case addAction(a):
    if a exists in CSP then
      Remove constraints a[i] = 0 for all 0 ≤ i < k
    else
      Create new CSP model including new action description a
  case newPlan(goal):
    SET_GOAL_CONSTRAINT(goal)
    return SOLVE_CSP
  case monitor(plan, s):
    A := GETNEXT_PARALLEL_ACTS(plan, s)
    F := EXECUTE_PAR_ACTS(A, s) // Form futures sequence
    for f[a, serv] ← F do
      // f concerning a and service instance serv
      ON_FUTURE_COMPLETE(f, a, s)
    end for
    F onAllComplete
      // short timeout has expired or all futures responded
      CHECK_PLAN(plan, s)
end function

function ON_FUTURE_COMPLETE(f, a, s)
  f onComplete    // Future return or timeout: run in parallel
    case success:
      if response contains sensed output then
        {UPDATE_CONTEXT(out)}
        if persistent info then
          {BOOKKEEPBEHAVIOR(f, inParams(f, s), out)}
      if f ∈ pendingActors then    // f with long timeout returned
        Add f to pendingActors
    case short timeout:
      if service(f) has justifiably longTimeout then
        // f in pendingActors until response or longTimeout expires
        Add f to pendingActors
        // Proceed as if a's effects have been materialized
        {UPDATE_CONTEXT(effectVars(a)[succ(s)])}
        BOOKKEEPVALUES(f, effectVars(a)[s])
    case failure:
      if f ∈ pendingActors then    // f with long timeout returned
        Remove f from pendingActors
      BOOKKEEPBEHAVIOR(f, inParams(f, s), failure)
end function

function EXECUTE_PAR_ACTS(A, s)    // Returns a set of futures F
  for pf ← pendingActors do
    if longTimeout(pf) has expired then
      Remove pf from pendingActors
      UNDO_EFFECTS(pf)
      CHECK_PLAN(plan, s); return
    else
      F.add(pf)
  end for
  for a ← A do
    if a not dependent on any other action in pendingActors AND
     not scheduled before at s then
      serv := GET_NEXT_INSTANCE(a, inParams(a, s))
      F.add(EXECUTE(serv, inParams(a, s)))
    if a has been scheduled before at s then
      // Detected a malicious action
      BOOKKEEPBEHAVIOR(f, inParams(f, s), failure)
  end for
  return F
end function
```

```
function CHECK_PLAN(plan, s)
    // Partial assignment at solver level, complete solution
    updPlan := REFINE_PLAN(plan, s)
    if updPlan ≠ ∅ then
        send monitor(updPlan, 0)
    else    // Refinement failed, replan from scratch
        newPlan :=REPLAN
        if newPlan ≠ ∅ then
            send monitor(newPlan, 0)
        else
            notify client "Goal is not satisfiable"
end function

function REFINE_PLAN(plan, s)
    // Instantiation of known variables at the solver level
    COPY_CONTEXT_TO_INIT_STATE
    planSuffix := CONSISTENT(plan, s)
    if empty(planSuffix) then
        // CSP-level action variables in plan suffix after s are set to 1
        // Input parameters and rest of action variables left unassigned
        ASSIGN_PARTIAL_SOLUTION(plan, s)
        return SOLVE_CSP
    else
        return x6planSuffix
end function
```

References

[1] M. Aiello, M. Papazoglou, J. Yang, M. Carman, M. Pistore, L. Serafini, P. Traverso, A request language for web-services based on planning and constraint satisfaction, in: VLDB Workshop on Technologies for E-Services (TES), in: LNCS, Springer, 2002, pp. 76–85.

[2] A. Albore, H. Palacios, H. Geffner, A translation-based approach to contingent planning, in: Proc. of the 21st International Joint Conference on Artificial Intelligence (IJCAI), 2009, pp. 1623–1628.

[3] A. Albore, M. Ramírez, H. Geffner, Effective heuristics and belief tracking for planning with incomplete information, in: Proc. of the 21st International Conference on Automated Planning and Scheduling (ICAPS), AAAI, 2011.

[4] T. Au, U. Kuter, D. Nau, Web service composition with volatile information, in: 4th International Semantic Web Conference (ISWC), Springer, 2005, pp. 52–66.

[5] F. Baligand, N. Rivierre, T. Ledoux, A declarative approach for QoS-aware web service compositions, in: Proc. of the 5th International Conference in Service-Oriented Computing (ICSOC), AAAI, 2007, pp. 422–428.

[6] R. Barták, A novel constraint model for parallel planning, in: Proc. of the 24th International Florida Artificial Intelligence Research Society Conference (FLAIRS), 2011.

[7] R. Barták, M.A. Salido, F. Rossi, New trends in constraint satisfaction, planning, and scheduling: a survey, Knowl. Eng. Rev. 25 (3) (2010) 249–279.

[8] R. Barták, D. Toropila, Reformulating constraint models for classical planning, in: Proc. of the 21st International Florida Artificial Intelligence Research Society Conference (FLAIRS), AAAI, 2008, pp. 525–530.

[9] R. Barták, D. Toropila, Enhancing constraint models for planning problems, in: Proc. of the 22nd International Florida Artificial Intelligence Research Society Conference (FLAIRS), AAAI, 2009.

[10] J.C. Beck, M.S. Fox, Constraint-directed techniques for scheduling alternative activities, Artif. Intell. 121 (1–2) (2000) 211–250.

[11] E. Benazera, Alternatives to re-planning: methods for plan re-evaluation at runtime, in: Workshop on Plan Execution: A Reality Check, 15th International Conference on Automated Planning and Scheduling (ICAPS), 2005.

[12] D. Berardi, D. Calvanese, G. De Giacomo, R. Hull, M. Mecella, Automatic composition of transition-based semantic web services with messaging, in: Proc. of the 31st International Conference on Very Large Data Bases, 2005, pp. 613–624.

[13] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, M. Mecella, Automatic composition of e-services that export their behavior, in: Proc. of the 1st International Conference on Service-Oriented Computing (ICSOC), Springer, 2003, pp. 43–58.

[14] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, M. Mecella, Automatic service composition based on behavioral descriptions, Int. J. Coop. Inf. Syst. 14 (4) (2005) 333–376.

[15] D. Berardi, F. Cheikh, G. De Giacomo, F. Patrizi, Automatic service composition via simulation, Int. J. Found. Comput. Sci. 19 (2) (2008) 429–451.

[16] D. Berardi, G. De Giacomo, M. Lenzerini, M. Mecella, D. Calvanese, Synthesis of underspecified composite *e*-services based on automated reasoning, in: Proc. of the 2nd International Conference on Service-Oriented Computing, Springer, 2004, pp. 105–114.

[17] P. Bertoli, R. Kazhamiakin, M. Paolucci, M. Pistore, H. Raik, M. Wagner, Continuous orchestration of web services via planning, in: Proc. of the 19th International Conference on Automated Planning and Scheduling (ICAPS), AAAI, 2009.

[18] P. Bertoli, M. Pistore, P. Traverso, Automated web service composition by on-the-fly belief space search, in: Proc. of the 16th International Conference on Automated Planning and Scheduling (ICAPS), AAAI, 2006, pp. 358–361.

[19] P. Bertoli, M. Pistore, P. Traverso, Automated composition of web services via planning in asynchronous domains, Artif. Intell. 174 (2010) 316–361.

[20] J. Blythe, Planning under uncertainty in dynamic domains, PhD thesis, Carnegie Mellon University, 1998.

[21] D. Borrajo, M. Veloso, Probabilistically reusing plans in deterministic planning, in: Proc. of ICAPS-12 Workshop on Heuristics and Search for Domain-Independent Planning, 2012, pp. 17–25.

[22] R.I. Brafman, J. Hoffmann, Conformant planning via heuristic forward search: a new approach, in: Proc. of the 14th International Conference on Automated Planning and Scheduling (ICAPS), 2004, pp. 355–364.

[23] M. Brenner, B. Nebel, Continual planning and acting in dynamic multiagent environments, Auton. Agents Multi-Agent Syst. 19 (3) (2009) 297–331.

[24] M. Caruso, C.D. Ciccio, E. Iacomussi, E. Kaldeli, A. Lazovik, M. Mecella, Service ecologies for home/building automation, in: Proc. of the 10th IFAC Symposium on Robot Control, 2012.

[25] Choco, Choco library documentation, www.emn.fr/z-info/choco-solver, 2012.

[26] G. Chu, S. Gaspers, N. Narodytska, A. Schutt, T. Walsh, On the complexity of global scheduling constraints under structural restrictions, in: Proc. of the 23rd International Joint Conference on Artificial Intelligence (IJCAI), 2013.

[27] A. Cimatti, M. Pistore, M. Roveri, P. Traverso, Weak, strong, and strong cyclic planning via symbolic model checking, Artif. Intell. 147 (1–2) (2003) 35–84.
[28] G. De Giacomo, F. Patrizi, S. Sardiña, Automatic behavior composition synthesis, Artif. Intell. 196 (2013) 106–142.
[29] M.B. Do, S. Kambhampati, Planning as constraint satisfaction: solving the planning-graph by compiling it into CSP, Artif. Intell. 132 (2001) 151–182.
[30] S. Duan, S. Babu, K. Munagala, Fa: a system for automating failure diagnosis, in: Proc. of the 25th International Conference on Data Engineering (ICDE), 2009, pp. 1012–1023.
[31] J. Fan, S. Kambhampati, A snapshot of public web services, SIGMOD Rec. 34 (1) (2005) 24–32.
[32] H. Fargier, J. Lang, J.M. Lang, T. Schiex, Mixed constraint satisfaction: a framework for decision problems under incomplete knowledge, in: Proc. of the 13th AAAI Conference on Artificial Intelligence, 1996, pp. 175–180.
[33] J. Finger, Exploiting constraints in design synthesis, PhD thesis, Stanford University, 1987.
[34] J. Foss, N. Onder, D. Smith, Preventing unrecoverable failures through precautionary planning, in: Proc. of the 10th Workshop on Moving Planning and Scheduling Systems into the Real World, 2007.
[35] M. Fox, A. Gerevini, D. Long, I. Serina, Plan stability: replanning versus plan repair, in: Proc. of the 16th International Conference on Automated Planning and Scheduling, 2006, pp. 212–221.
[36] M. Fox, D. Long, Pddl2.1: an extension to PDDL for expressing temporal planning domains, J. Artif. Intell. Res. 20 (2003) 61–124.
[37] I. Georgievski, Hierarchical planning definition language, Technical report, University of Groningen, 2013, JBI 2013-12-3.
[38] A. Gerevini, P. Haslum, D. Long, A. Saetti, Y. Dimopoulos, Deterministic planning in the fifth international planning competition: PDDL3 and experimental evaluation of the planners, Artif. Intell. 173 (5–6) (2009) 619–668.
[39] A. Gerevini, D. Long, Preferences and soft constraints in PDDL3, in: 16th ICAPS Workshop on Planning with Preferences and Soft Constraints, 2006.
[40] A. Gerevini, A. Saetti, I. Serina, An approach to temporal planning and scheduling in domains with predictable exogenous events, J. Artif. Intell. Res. 25 (1) (2006) 187–231.
[41] M. Ghallab, H. Laruelle, Representation and control in IxTeT, a temporal planner, in: Proc. of the 2nd International Conference on Artificial Intelligence Planning Systems, 1994.
[42] M. Ghallab, D. Nau, P. Traverso, Automated Planning: Theory and Practice, Morgan Kaufmann, Amsterdam, 2004.
[43] M. Göbelbecker, C. Gretton, R. Dearden, A switching planner for combined task and observation planning, in: Proc. of the 25th AAAI Conference on Artificial Intelligence, AAAI, 2011.
[44] K. Golden, Leap before you look: information gathering in the PUCCINI planner, in: Proc. of the 6th International Conference on Artificial Intelligence Planning Systems (AIPS), 1998, pp. 70–77.
[45] K. Golden, A domain description language for data processing, in: ICAPS Workshop on the Future of PDDL, AAAI, 2003.
[46] K. Golden, O. Etzioni, D. Weld, Planning with execution and incomplete information, Technical Report 97-11-05, UW CSE, 1996.
[47] K. Golden, W. Pang, A constraint-based planner applied to data processing domains, in: Proc. of the 10th International Conference on Principles and Practice of Constraint Programming (CP), 2004, p. 815.
[48] K. Golden, D.S. Weld, Representing sensing actions: the middle ground revisited, in: Proc. of the 5th International Conference on Principles of Knowledge Representation and Reasoning (KR), 1996, pp. 174–185.
[49] I. Greif, Semantics of communicating parallel processes, PhD thesis, MIT, 1975.
[50] C. Guettier, N. Yorke-Smith, Enhancing the anytime behaviour of mixed CSP-based planning, in: Proc. of ICAPS Workshop on Planning Under Uncertainty for Autonomous Systems, 2005, pp. 29–38.
[51] A.B. Hassine, S. Matsubara, T. Ishida, A constraint-based approach to horizontal web, in: Proc. of the 5th International Semantic Web Conference (ISWC2006), 2006, pp. 130–143.
[52] M. Helmert, The fast downward planning system, J. Artif. Intell. Res. 26 (2006) 191–246.
[53] M. Helmert, Concise finite-domain representations for PDDL planning tasks, Artif. Intell. 173 (2009) 503–535.
[54] C. Hewitt, P. Bishop, R. Steiger, A universal modular actor formalism for artificial intelligence, in: Proc. of the 3rd International Joint Conference on Artificial Intelligence (IJCAI), 1973.
[55] J. Hoffmann, Towards efficient belief update for planning-based web service composition, in: 18th European Conference on Artificial Intelligence (ECAI), 2008, pp. 558–562.
[56] J. Hoffmann, P. Bertoli, M. Helmert, M. Pistore, Message-based web service composition, integrity constraints, and planning under uncertainty: a new connection, J. Artif. Intell. Res. 35 (2009) 49–117.
[57] J. Hoffmann, P. Bertoli, M. Pistore, Web service composition as planning, revisited: in between background theories and initial state uncertainty, in: Proc. of the 21st AAAI Conference on Artificial Intelligence, AAAI, 2007, pp. 1013–1018.
[58] J. Hoffmann, R.I. Brafman, Contingent planning via heuristic forward search with implicit belief states, in: Proc. of the 15th International Conference on Automated Planning and Scheduling (ICAPS), AAAI, 2005, pp. 71–80.
[59] J. Hoffmann, B. Nebel, The FF planning system: fast plan generation through heuristic search, J. Artif. Intell. Res. 14 (2001) 253–302.
[60] J. Hoffmann, I. Weber, F. Kraft, SAP speaks PDDL, in: Proc. of the 4th AAAI Conference on Artificial Intelligence, AAAI, 2010.
[61] N. Hyafil, F. Bacchus, Conformant probabilistic planning via CSPs, in: Proc. of the 13th International Conference on Automated Planning and Scheduling (ICAPS), 2003, pp. 205–214.
[62] N. Hyafil, F. Bacchus, Utilizing structured representations and CSPs in conformant probabilistic planning, in: Proc. of the 16th Eureopean Conference on Artificial Intelligence (ECAI), 2004, pp. 1033–1034.
[63] L. Iocchi, T. Lukasiewicz, D. Nardi, R. Rosati, Reasoning about actions with sensing under qualitative and probabilistic uncertainty, ACM Trans. Comput. Log. 10 (1) (2009).
[64] F. Kabanza, M. Barbeau, R. St-Denis, Planning control rules for reactive agents, Artif. Intell. 95 (1997) 67–113.
[65] E. Kaldeli, Domain-independent planning for services in uncertain and dynamic environments, PhD thesis, University of Groningen, 2013.
[66] E. Kaldeli, A. Lazovik, M. Aiello, Extended goals for composing services, in: Proc. of the 19th International Conference on Automated Planning and Scheduling (ICAPS 2009), AAAI, 2009.
[67] E. Kaldeli, A. Lazovik, M. Aiello, Continual planning with sensing for web service composition, in: Proc. of the 25th AAAI Conference on Artificial Intelligence, AAAI, 2011.
[68] E. Kaldeli, E. Warriach, A. Lazovik, M. Aiello, Coordinating the web of services for a smart home, ACM Trans. Web 7 (2) (2013) 10.
[69] E. Kaldeli, E.U. Warriach, J. Bresser, A. Lazovik, M. Aiello, Interoperation, composition and simulation of services at home, in: 8th International Conference on Service Oriented Computing (ICSOC), in: LNCS, vol. 6470, Springer, 2010, pp. 167–181.
[70] S. Kambhampati, Planning graph as a (dynamic) CSP: exploiting EBL, DDB and other CSP search techniques in GraphPlan, J. Artif. Intell. Res. 12 (2000) 1–34.
[71] M. Klusch, A. Gerber, Semantic web service composition planning with OWLS-Xplan, in: Proc. of the 1st International AAAI Fall Symposium on Agents and the Semantic Web, 2005, pp. 55–62.
[72] M. Klusch, A. Gerber, Fast composition planning of OWL-S services and application, in: Proc of the 4th IEEE European Conference on Web Services (ECOWS), 2006, pp. 181–190.

[73] M. Klusch, K.-U. Renner, Fast dynamic re-planning of composite OWL-S services, in: Proc. of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT) – Workshops, 2006, pp. 134–137.
[74] C.A. Knoblock, Planning, executing, sensing, and replanning for information gathering, in: International Joint Conference of Artificial Intelligence (IJCAI), Morgan Kaufmann, 1995, pp. 1686–1693.
[75] C. Krause, Z. Maraikar, A. Lazovik, F. Arbab, Modeling dynamic reconfigurations in Reo using high-level replacement systems, Sci. Comput. Program. 76 (1) (2011) 23–36.
[76] U. Kuter, E. Sirin, B. Parsia, D.S. Nau, J.A. Hendler, Information gathering during planning for web service composition, J. Web Semant. 3 (2–3) (2005) 183–205.
[77] P. Laborie, Algorithms for propagating resource constraints in AI planning and scheduling: existing approaches and new results, Artif. Intell. 143 (2) (2003) 151–188.
[78] A. Lazovik, M. Aiello, R. Gennari, Encoding requests to web service compositions as constraints, in: 11th International Conference on Principles and Practice of Constraint Programming (CP), Springer, 2005.
[79] A. Lazovik, M. Aiello, M. Papazoglou, Planning and monitoring the execution of web service requests, in: Proc. of the 1st International Conference on Service-Oriented Computing (ICSOC), Springer, 2003, pp. 335–350.
[80] A. Lazovik, M. Aiello, M. Papazoglou, Planning and monitoring the execution of web service requests, Int. J. Digit. Libr. 6 (3) (2006) 235–246.
[81] A. Lazovik, F. Arbab, Using Reo for service coordination, in: Proc. of the 5th International Conference in Service-Oriented Computing (ICSOC), Springer, 2007, pp. 398–403.
[82] C. Lecoutre, Constraint Networks: Targeting Simplicity for Techniques and Algorithms, Wiley–ISTE, 2009.
[83] N. Lin, U. Kuter, E. Sirin, Web service composition with user preferences, in: Proc. of the 5th European Semantic Web Conference (ESWC), Springer, 2008, pp. 629–643.
[84] A. Lopez, F. Bacchus, Generalizing GraphPlan by formulating planning as a CSP, in: Proc. of the 18th International Joint Conference on Artificial Intelligence (IJCAI), Morgan Kaufmann, 2003, pp. 954–960.
[85] E. Martínez, Y. Lespérance, Web service composition as a planning task: experiments using knowledge-based planning, in: Proc. of the ICAPS Workshop on Planning and Scheduling for Web and Grid Services, AAAI, 2004.
[86] F.I. Matthieu Gallien, S. Lemai, Robot actions planning and execution control for autonomous exploration rovers, in: Workshop on Plan Execution: A Reality Check, 15th International Conference on Automated Planning and Scheduling (ICAPS), 2005.
[87] D. McDermott, AIPS-98 Planning Competition Committee, PDDL: the planning domain definition language, Technical Report CVC TRR98003/DCS TR1165, 1998.
[88] D.V. McDermott, Estimated-regression planning for interactions with web services, in: Proc. of the 6th International Conference on Artificial Intelligence Planning Systems (AIPS), AAAI, 2002, pp. 204–211.
[89] S. McIlraith, T.C. Son, Adapting Golog for composition of semantic web-services, in: D. Fensel, F. Giunchiglia, D. McGuinness, M. Williams (Eds.), Proc. of the 8th International Conference on Principles of Knowledge Representation (KR), Morgan Kaufmann, 2002, pp. 482–496.
[90] S.A. McIlraith, Towards declarative programming for web services, in: Proc. of the 11th International Symposium on Static Analysis (SAS), Springer, 2004, p. 21.
[91] S. Murugan, V. Ramachandran, Aspect oriented decision making model for byzantine agreement, J. Comput. Sci. 8 (2012) 382–388.
[92] D.J. Musliner, Using abstraction and nondeterminism to plan reaction loops, in: Proc. of the 12th National Conference on Artificial Intelligence (AAAI), AAAI, 1994, pp. 1036–1041.
[93] D.J. Musliner, E.H. Durfee, K.G. Shin, World modeling for the dynamic construction of real-time control plans, Artif. Intell. 74 (1995) 83–127.
[94] B. Nebel, Y. Dimopoulos, J. Koehler, Ignoring irrelevant facts and operators in plan generation, in: Proc. of the 4th European Conference on Planning, 1997, pp. 338–350.
[95] B. Nebel, J. Koehler, Plan reuse versus plan generation: a theoretical and empirical analysis, Artif. Intell. 76 (1–2) (1995) 427–454.
[96] H. Palacios, H. Geffner, Compiling uncertainty away in conformant planning problems with bounded width, J. Artif. Intell. Res. 35 (2009) 623–675.
[97] J. Peer, A PDDL based tool for automatic web service composition, in: Proc. of the 2nd International Workshop on the Principles and Practice of Semantic Web Reasoning, in: Lecture Notes in Computer Science, vol. 3208, Springer, 2004.
[98] J. Peer, A POP-based replanning agent for automatic web service composition, in: Proc. of the 2nd European Semantic Web Conference (ESWC), Springer, 2005, pp. 47–61.
[99] R.P.A. Petrick, An extension of knowledge-level planning to interval-valued functions, in: AAAI Workshop on Generalized Planning, 2011.
[100] R.P.A. Petrick, F. Bacchus, Extending the knowledge-based approach to planning with incomplete information and sensing, in: Proc. of the 14th International Conference on Automated Planning and Scheduling (ICAPS), 2004, pp. 2–11.
[101] T. Pilioura, A. Tsalgatidou, Unified publication and discovery of semantic web services, ACM Trans. Web 3 (3) (2009) 11.
[102] M. Pistore, A. Marconi, P. Bertoli, P. Traverso, Automated composition of web services by planning at the knowledge level, in: 19th International Joint Conference on Artificial Intelligence, 2005, pp. 1252–1259.
[103] M. Pistore, L. Spalazzi, P. Traverso, A minimalist approach to semantic annotations for web processes compositions, in: Proc. of the 3rd European Semantic Web Conference (ESWC), 2006, pp. 620–634.
[104] M. Pistore, P. Traverso, P. Bertoli, Automated composition of web services by planning in asynchronous domains, in: Proc. of the 15th International Conference on Automated Planning and Scheduling (ICAPS), AAAI, 2005, pp. 2–11.
[105] G. Shani, R.I. Brafman, Replanning in domains with partial information and sensing actions, in: Proc. of the 22nd International Joint Conference on Artificial Intelligence (IJCAI), IJCAI/AAAI, 2011, pp. 2021–2026.
[106] D. Shaparau, M. Pistore, P. Traverso, Fusing procedural and declarative planning goals for nondeterministic domains, in: Proc. of the 23rd AAAI Conference on Artificial Intelligence, AAAI, 2008, pp. 983–990.
[107] M. Sheshagiri, M. DesJardins, T. Finin, A planner for composing services described in DAML-S, in: Proc. of the 13th ICAPS Workshop on Planning for Web Services, 2003.
[108] E. Sirin, B. Parsia, D. Wu, J.A. Hendler, D.S. Nau, HTN planning for web service composition using shop2, J. Web Semant. 1 (4) (2004) 377–396.
[109] D. Skoutas, D. Sacharidis, A. Simitsis, T. Sellis, Serving the Sky: discovering and selecting semantic web services through dynamic skyline queries, in: 2nd IEEE International Conference on Semantic Computing, 2008, pp. 222–229.
[110] S. Sohrabi, N. Prokoshyna, S.A. McIlraith, Web service composition via generic procedures and customizing user preferences, in: Proc. of the 5th International Semantic Web Conference (ISWC), Springer, 2006, pp. 597–611.
[111] S. Sohrabi, N. Prokoshyna, S.A. McIlraith, Web service composition via the customization of Golog programs with user preferences, in: Conceptual Modeling: Foundations and Applications, vol. 5600, Springer, 2009, pp. 319–334.
[112] S.T. To, T.C. Son, E. Pontelli, Contingent planning as and/or forward search with disjunctive representation, in: Proc. of the 21st International Conference on Automated Planning and Scheduling (ICAPS), AAAI, 2011.
[113] P. Traverso, M. Pistore, Automated composition of semantic web services into executable processes, in: Proc. of the 3rd International Semantic Web Conference, Springer, 2004, pp. 380–394.
[114] N. van Beest, E. Kaldeli, P. Bulanov, J. Wortmann, A. Lazovik, Automated runtime repair of business processes, Inf. Sci. 39 (2014) 45–79.

[115] R. van der Krogt, M. de Weerdt, Plan repair as an extension of planning, in: Proc. of the 15th International Conference on Automated Planning and Scheduling, AAAI, 2005, pp. 161–170.
[116] V. Vidal, Branching and pruning: an optimal temporal POCL planner based on constraint programming, Artif. Intell. 170 (2004) 570–577.
[117] R.J. Wallace, D. Grimes, Problem-structure vs. solution-based methods for solving dynamic constraint satisfaction problems, in: Proc. of the 22nd International Conference on Tools with Artificial Intelligence, 2010.
[118] R.J. Wallace, D. Grimes, E.C. Freuder, Solving dynamic constraint satisfaction problems by identifying stable features, in: Proc. of the 21st International Joint Conference on Artificial Intelligence, 2009, pp. 621–627.
[119] E.U. Warriach, E. Kaldeli, J. Bresser, A. Lazovik, M. Aiello, A tool for integrating pervasive services and simulating their composition, in: Demo Session of the 8th International Conference in Service-Oriented Computing (ICSOC), in: LNCS, Springer, 2010, pp. 726–727.
[120] K. Westra, Web service composition: connecting the web cloud, Bachelor's thesis, University of Groningen, 2010.
[121] D. Wu, B. Parsia, E. Sirin, J.A. Hendler, D.S. Nau, Automating DAML-S web services composition using SHOP2, in: Proc. of the 2nd International Semantic Web Conference, 2003, pp. 195–210.
[122] S.W. Yoon, A. Fern, R. Givan, FF-Replan: a baseline for probabilistic planning, in: Proc. of the 17th International Conference on Automated Planning and Scheduling, AAAI, 2007, p. 352.
[123] S.W. Yoon, A. Fern, R. Givan, S. Kambhampati, Probabilistic planning via determinization in hindsight, in: Proc. of the 23rd AAAI Conference on Artificial Intelligence, 2008, pp. 1010–1016.
[124] T. Yu, Y. Zhang, K.-J. Lin, Efficient algorithms for web services selection with end-to-end QoS constraints, ACM Trans. Web 1 (2007).
[125] S. Yumatov, Web-based interface for a smart home, Bachelor thesis, 2011.