

Structure Constraints in a Constraint-Based Planner

Wanlin Pang* and Keith Golden

NASA Ames Research Center
Moffett Field, CA 94035
{wpang | kgolden}@email.arc.nasa.gov

Abstract. In this paper we report our work on a new constraint domain, where variables can take structured values. Earth-science data processing (ESDP) is a planning domain that requires the ability to represent and reason about complex constraints over structured data, such as satellite images. This paper reports on a constraint-based planner for ESDP and similar domains. We discuss our approach for translating a planning problem into a constraint satisfaction problem (CSP) and for representing and reasoning about structured objects and constraints over structures.

1 Introduction

Earth-science data processing (ESDP) at NASA is the problem of transforming low-level observations of the Earth system, such as data from Earth-observing satellites and ground weather stations, into high-level observations or predictions, such as crop failure or high fire risk. Given the large number of socially and economically important variables that can be derived from the data, the complexity of the data processing needed to derive them and the many terabytes of data that must be processed each day, there are great challenges and opportunities in processing the data in a timely manner, and a need for more effective automation. Our approach to providing this automation is to cast it as a planning problem: we represent data-processing operations as planner actions and desired data products as planner goals, and use a planner to generate data-flow programs that produce the requested data.

Many of the recent advances in planning, such as state-based heuristic search or reduction to satisfiability problems, are not readily adapted to ESDP problems, due to the following features:

- **universal quantification:** Many commands and programs operate on sets of things, where membership in the set can be defined in terms of necessary and sufficient conditions. For example, the Unix *ls* command lists all files in a given directory.
- **incomplete information:** It is common for a planner to have only incomplete information at the time of planning. For example, a planner is unlikely to know about all the files on the local file system, until *ls* command is executed.
- **large, dynamic universe:** The size of the universe is generally very large or infinite. For example, there are hundreds of thousands of files accessible on a typical file

* QSS Group Inc.

system and billions of web pages over the Internet. The number of possible files, file pathnames, etc, is effectively infinite. Furthermore, new files can be created by executing actions, so the universe is dynamic.

- **complex data types:** Files and other objects in the domain are complex data structures, specified in terms of their attributes, which can, in turn, be complex data types. For example, a satellite image is specified by resolution, date, region, etc, the region can be specified by a pair of points defining its bounding box, and a point is a pair of coordinates designating the longitude and latitude.
- **complex constraints:** Data processing domain typically involves a rich set of constraints. For example, specifications of data inputs and outputs include constraints indicating geographic regions of interest, thresholds on resolution, data quality, file size, etc. Specifications of data-processing operations include constraints relating the inputs of the operations to the outputs, which are complex objects such as satellite images and weather forecast data. In the course of planning, additional constraints arise specifying how parameters of an action depend on the parameters of other actions in the plan.

We take the approach, like many other researchers [21,16,7,20], of translating the planning problem into a constraint satisfaction problem (CSP). However, since data processing domains are substantially different from other planning domains that have been explored, our approach to translating planning problems to CSPs differs as well. For example, [7] use variables to represent goals and domains to represent available planner actions achieving the goals. Constraints are used to encode mutual exclusion relations. While this is an effective approach for propositional planning problems, we also need variables to represent objects and action parameters, and constraints to represent relations among them. Thus, our encoding is somewhat more complex.

For example, actions can have inputs and outputs, which are represented as variables, typically of some complex (structured) type. Attributes of these inputs and outputs may also be referenced by variables, and constraints over any of these variables may be specified as part of the action description. For example, an action that produces a scaled-down copy of an image might have a constraint specifying that the resolution of the output image equals the resolution of the input times a scale factor (which is a parameter of the action). Other attributes of the input image, such as the subject matter, will be unchanged in the output.

In this paper, we report our recent work on representing and reasoning about structured objects, which we refer to as structures. Section 2 discusses data processing as a planning domain and our planning approach, focusing on how we handle structures. Section 3 discusses how we represent constraints over structures, and Section 4 describes our approach for solving constraint problems that include structures.

2 Planning in the data processing domain

2.1 TOPS – a data processing domain

The Terrestrial Observation and Prediction System (TOPS, <http://www.forestry.umd.edu/ntsg/Projects/TOPS/>)[18] is an ecological forecasting system that assimilates data

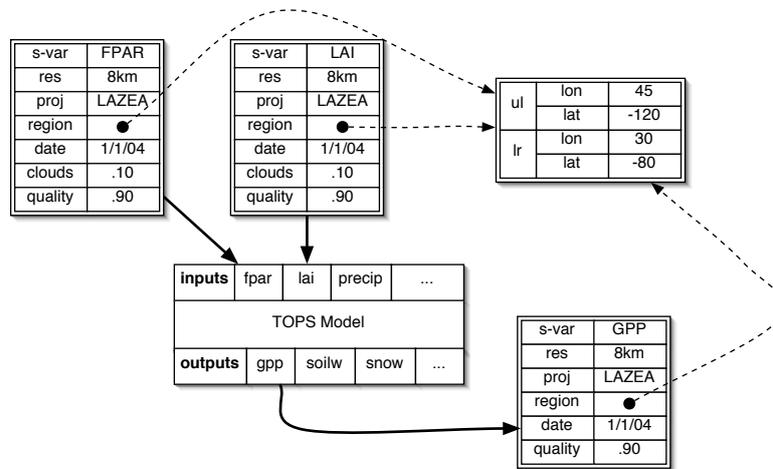


Fig. 1. Structured inputs and outputs to a model in the TOPS domain. Solid arrows represent data flow. Broken arrows represent sub-structure relations.

from Earth-orbiting satellites and ground weather stations to model and forecast conditions on the surface, such as soil moisture, vegetation growth and plant stress. The goal of this system is to monitor and predict changes in key environmental variables. The inputs needed by a TOPS model run include satellite data, such as Fractional Photosynthetically Active Radiation (FPAR), or Leaf Area Index (LAI) and weather data, such as precipitation. The input data may be obtained directly from data archives somewhere, but most of times, the data obtained from data sources need to be processed before the model run. A common sequence of data processing is: 1) gather data from multiple sources; 2) convert the data into a common representation, combine data, and perform other transformations; 3) feed the data into a model, for example, a simulation of gross primary production (GPP) of terrestrial vegetation; 4) convert the output of the model into some form suitable for visualization. Routine data processing can consume roughly 80% of manpower, with only 20% devoted to data analysis. As a result, the vast majority of data are never used, in part due to the effort required to prepare data [17]. We have developed a planner-based agent, called IMAGEbot, to automate these data-processing activities.

The data consumed and produced in the TOPS domain are all complex data structures, such as spatial data. Figure 1 shows a simplified view of the data input and output to a TOPS ecological model.

2.2 Planning approach

The architecture of IMAGEbot is described in Figure 2. Planning domains are specified in an expressive language called the Data Processing Action Description Language (DPADL) [9]. From the planning problems specified in DPADL, the planner incremen-

tally constructs a *lifted planning graph*, from which it extracts distance estimates for heuristic search and also derives a CSP representation of the planning problem. Whereas a conventional planning graph [3] is a grounded representation, consisting of ground actions and propositions, a *lifted* planning graph contains variables. This is a much more concise representation than an ordinary planning graph, but it is potentially less informative. We use a constraint propagation algorithm to restrict the domains of variables in the graph, making it more informative. The focus of this paper is on the CSP representation. The planning search and constraint propagation are conducted iteratively, until a plan is found or the planner proves that no valid plan exists in the planning graph, in which case, it either extends the planning graph, or admits failure.

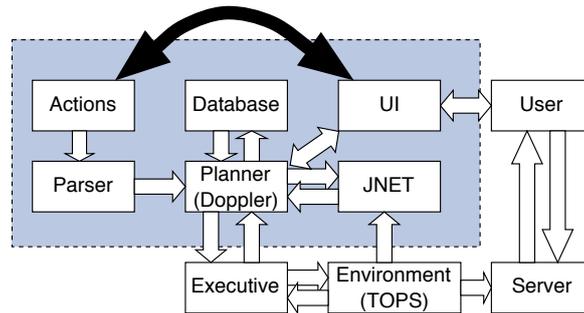


Fig. 2. The IMAGEbot architecture

Actions and conditions An action is a tuple $\langle I, O, \mathcal{P}, \Pi, \mathcal{E}, \chi \rangle$, where I, O, \mathcal{P} are the *input* variables, *output* variables and *parameters*, respectively. All these variables are typed. Π is the precondition, \mathcal{E} is a list of *effects* and χ is a procedure for executing the action that may reference any variable in $I \cup \mathcal{P}$ and must set every variable in O .

A full discussion of preconditions and effects in DPADL can be found in [9]; for the purposes of this paper, it suffices to observe that many goals and preconditions consist of requirements on the attributes of variables in I and many effect conditions consist of assignments to the attributes of variables in O and creation of new objects (which themselves are specified in terms of assignments on attributes). These conditions can be expressed in the concise canonical form $v = \langle a_1, a_2, \dots, a_n \rangle$, where v is a variable and $\langle a_1, a_2, \dots, a_n \rangle$ is a structure specification, where each attribute a_i may be a variable, constant, structure specification, or \emptyset (unspecified). For example, suppose the attributes of a file are name, format, and size. A file can be represented as a tuple $\langle \text{name}, \text{format}, \text{size} \rangle$. To specify the goal of finding a file f named “foo.txt” whose size is greater than 100, we could write $f = \langle \text{"foo.txt"}, \emptyset, s \rangle \wedge s > 100$.

Like goals, effects can also have \emptyset -attributes, but the meaning is different. In goals, \emptyset means *don't care*. In effects, it means *default*. Any variable $o \in O$ or new object may be specified as a *copy* of some variable $d \in I$, in which case attributes of o default to the same value as attributes of d . If nothing else were specified, then o would be a perfect copy of d . However, what we are interested in is typically not perfect copies, but

Algorithm 1 Structure Unification. \mathcal{E} and \mathcal{G} are structure specifications from an effect and goal, respectively. \mathcal{B} is the initial variable bindings and κ is a variable designating the object providing “default” values for \mathcal{E} .

$\text{unify}(\mathcal{E} = \langle e_1, \dots, e_k \rangle, \mathcal{G} = \langle g_1, \dots, g_k \rangle, \mathcal{B}, \kappa)$

1. **let** $\mathcal{D} := \emptyset$
 2. **unless**($\text{type}(\mathcal{E})$ super-type of $\text{type}(\mathcal{G})$)
 - (a) **return** \perp
 3. **for**($i = 1 \dots k$)
 - (a) **if**($e_i \neq \emptyset$)
 - i. **if**(g_i is a structure **and** e_i is a variable)
 - $\mathcal{B} := \text{unify}(\text{new Structure}(e_i), g_i, \mathcal{B}, e_i)$;
 - ii. **else if**(e_i is a structure **and** g_i is a variable)
 - **let** $\mathcal{N} := \text{new Structure}(g_i)$
 - $\mathcal{B} := \text{unify}(e_i, \mathcal{N}, \mathcal{B} \cup \{g_i \approx \mathcal{N}\}, \emptyset)$
 - iii. **else** $\mathcal{B} := \text{unify}(e_i, g_i, \mathcal{B})$
 - (b) **else**
 - i. **if**($g_i = \emptyset$) $g_i := \text{new variable}$
 - ii. **if**($\mathcal{D} = \emptyset$)
 - $\mathcal{D} := \text{new Structure}(\kappa)$ ($\mathcal{D} = \langle d_1 = \emptyset, \dots, d_i = \emptyset, \dots, d_k = \emptyset \rangle$)
 - $\mathcal{B} := \mathcal{B} \cup \{\kappa \approx \mathcal{D}\}$
 - iii. **if**($d_i = \emptyset$) $d_i := g_i$
 - iv. **else** $\mathcal{B} := \mathcal{B} \cup \{d_i \approx g_i\}$
 - (c) **else return** \perp
-

imperfect ones. For example, there are many actions that change just one or two properties of an object, such as file format, projection, resolution, size, or name. Specifying the outputs of those actions as copies of their inputs allows us to list only the attributes that are changed [12]. In our canonical form, an effect that changed only one attribute of o would be of the form $o = \langle \emptyset, \emptyset, \dots, n, \dots, \emptyset, \emptyset \rangle$, where n is the new value for the attribute that changed. All other attributes take on the corresponding value from d .

Structure unification Given a goal and effect in canonical form, determining whether (and under what conditions) the effect satisfies the goal is a simple matter of unification. Pure constraints, such as $s > 100$ in the example above, can never appear in effects, so they are just added to the CSP. Subgoals of the form $v = \langle a_1, a_2, \dots, a_n \rangle$ are matched against effects of the same form, using Algorithm 1. For the most part, this unification is just term-by-term matching, but \emptyset entries in effects, when matched with non- \emptyset entries in goals, result in delegation to whatever input variable to output is a copy of. This delegation results in an equality constraint between the the input variable and a new structure specification, which is treated as a new subgoal (Figure 3).

Lifted planning graphs The planner incrementally constructs a directed graph, similar to a planning graph [3], but using a lifted representation (*i.e.*, containing variables). This

graph is used to obtain distance estimates for heuristic search, and is also the basis for the construction of the CSP. Arcs in the graph are analogous two causal links [19]. A causal link is triple $\langle \alpha_s, p, \alpha_p \rangle$, recording the decision to use action α_s to support precondition p of action α_p . However, instead of an arc to record a commitment of support, we use it to indicate the *possibility* that α_s supports p . The lifted graph contains multiple ways of supporting p ; the choice of the actual supporter becomes a constraint satisfaction problem. We add an extra term to the arc for bookkeeping purposes – the condition $\gamma_p^{\alpha_s}$ needed in order for α_s to achieve p . A link then becomes $\langle \alpha_s, \gamma_p^{\alpha_s}, p, \alpha_p \rangle$.

Given an unsupported precondition p of action α_p , our first task is to identify all the actions that could support p . Because the universe is large and dynamic, identifying all possible ground actions that could support p would be impractical, so instead we use a lifted representation, identifying all action *schemas* that could provide support. Given an action schema α , we determine whether it supports p by *regressing* p through α_s (Figure 3). The result of regression is a formula $\gamma_p^{\alpha_s}$. If $\gamma_p^{\alpha_s} = \perp$, then α_s does not support p . The procedure for goal regression is straightforward, relying mainly on a unification-based entailment test, as described in Algorithm 1. Initial graph construction terminates when all preconditions have support or (more likely) a potential loop is detected.

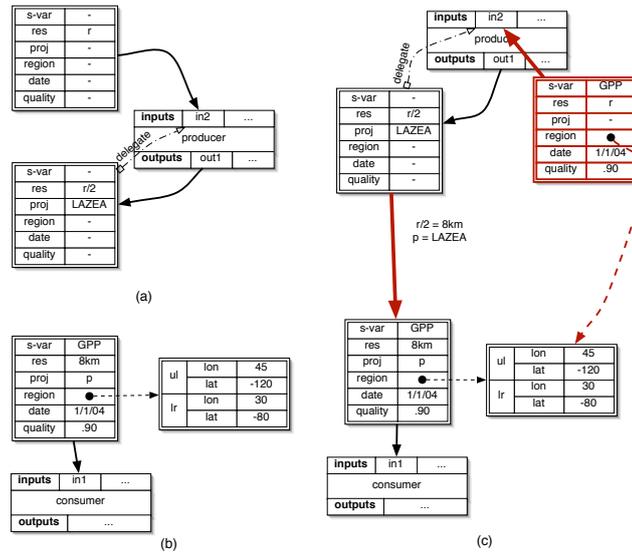


Fig. 3. Goal regression on structures. Both outputs (1) and inputs (2) of actions can be described as partial structure specifications (shown as double-bordered boxes). When matching an input (in1) to an output (out1) during planning, these structures are unified (c), resulting in equality constraints among attributes (labels on arc). Since the output out1 is defined as a copy of in2, goal conditions on attributes of in1 that are undefined for out1 are delegated to in2, resulting in a new subgoal.

From planning to constraints After the planning graph is constructed, a constraint satisfaction problem (CSP) representing the search space is incrementally built. The CSP contains: 1) boolean variables for all arcs, nodes and conditions; 2) variables for all parameters, input and output variables and function values; 3) for every condition in the graph, a constraint specifying when that condition holds (for conditions supported by links, this is just the XOR of the arc variables) ; 4) for conjunctive and disjunctive expressions, the constraint is the respective conjunction or disjunction of the boolean variables corresponding to appropriate sub-expressions; 5) for every arc in the graph, constraints specifying the conditions under which the supported fluents will be achieved (i.e., $\gamma_p^\alpha \Rightarrow p$); 6) user-specified constraints; and 7) constraints representing structured objects.

3 Constraints over structures

We start by reviewing some standard CSP notation and then describe how we represent and reason about structures.

3.1 Constraint satisfaction problems

A Constraint Satisfaction Problem (CSP) consists of variables, domains, and constraints. Formally, it can be defined as a triple $\langle X, D, C \rangle$ where $X = \{x_1, x_2, \dots, x_n\}$ is a finite set of variables, $D = \{d(x_1), d(x_2), \dots, d(x_n)\}$ is a set of domains containing values the variables may take, and $C = \{C_1, C_2, \dots, C_m\}$ is a set of constraints. Each constraint C_i is defined as a relation R on a subset of variables $V = \{x_i, x_j, \dots, x_k\}$, called the constraint scope. R may be represented extensionally as a subset of the Cartesian product $d(x_i) \times d(x_j) \times \dots \times d(x_k)$, or implicitly using a constraint procedure [13]. A constraint $C_i = (V_i, R_i)$ limits the values the variables in V can take simultaneously to those assignments that satisfy R . A solution to the problem is an assignment of values to variables in X satisfying constraints in C . The central reasoning task (or the task of solving a CSP) is to find one or more solutions.

In addition, the CSP converted from the planning problem, as discussed in Section 2.2 has the following features:

- Dynamics: variables and constraints may be dynamically added or removed from the problem, and values may be dynamically added or removed from domains.
- Infinity: variables may have unknown or infinite domains, making it impossible to represent constraints extensionally as relation tables.
- Typed variables: variables may take different types of values, such as numbers, booleans, strings and structured objects.
- Incompleteness: a solution to a problem doesn't require a complete assignment to all the variables in the problem, in part due to dynamics.

It has been reported in [8,10,11] how dynamics, infinity, string variables and constraints are handled. In this paper, we focus on how to represent and reason about structured objects, or structures for short. We call variables that take structures as values *structure variables*.

3.2 Structures

Data objects often contain complex data structures¹. Being able to efficiently represent and reason about these structures, as well as actions that create, copy or modify them, is essential. As discussed in Section 2.2, a structure can be specified in the concise canonical form $\langle a_1, a_2, \dots, a_n \rangle$, where a_i is attribute, which could be another structure specification. In the CSP, it also seems natural to represent a structured object as a tuple. For example, suppose a map has 3 attributes, *resolution*, *region*, and *date*. The tuple $\langle 8, \text{USA}, 2003154 \rangle$ represents a map of the US taken on the day 154 of year 2003 with resolution of 8 kilometers (one pixel corresponds to a 8km square). If a single object can be represented as a tuple, a set of objects of a given type can be represented as a relation table, where each row of the table is an object. If the structure attributes are represented as constraint variables (they are variables in the DPADL representation), a structure can be represented as a constraint, which is a relation by definition. Thus, it would seem unnecessary to have variables that take structures as values.

Unfortunately, this representation is limited to the situation where all the structure instances of a given type are known, there are only a finite number of them, and there are no constraints on structures but only on structure attributes. If the structure instances are unknown or there are infinitely many of them, which is more likely in real-world applications, there will be no finite representation of the constraints; if there are constraints on structures, we will end up with a CSP having 2nd-order constraints; that is, constraints over constraints.

We take the approach of representing structures as constraints, but in a different form. We allow the values of variables to be structures. As mentioned before, these variables are called structure variables. Like other variables, structure variables can appear in any form of constraints, and can be treated just as regular variables. Attributes of structures are also variables (which will be addressed as *attribute variables* if necessary). However, a structure variable also appears in a special form of constraint, called *structure constraint*. For each structure variable, we create a structure constraint on this structure variable and its attribute variables. For example, if we have a structure variable, x , that takes maps as values, we will have a structure constraint on $\langle x, x.res, x.region, x.time \rangle$, where $x.res$, $x.region$, and $x.time$ are attribute variables of x .

Although a structure constraint on a structure variable represents a set of structured objects, the representation is often implicit; it relates a structure variable to the structure attributes, capturing certain conditions relevant to structures in the DPADL coding of the planning problem. Adding structure variables into the CSP coding is not redundant but necessary to represent constraints over the structures. Now the question becomes how to specify structure constraints.

A structure constraint, like other constraints in our constraint library, is implemented as a procedure, or actually a collection of procedures over subsets of the structure and attribute variables. The applicability of these procedures depends on the domains of the structure and attributes variables, especially on whether they are finite or infinite.

¹ There are recursive structured objects such as filesystem directories and non-structured or semi-structured data objects such as the content of an image or text file, which are discussed in [11]. Here we limit the discussion to non-recursive structures.

Algorithm 3 Structure Constraint Procedure

$execute(x, \langle a_1, a_2, \dots, a_k \rangle, M_x, M_A, A)$

1. **if** $d(x)$ is finite
 - (a) for each $v \in d(x)$
 - i. **if** M_x is not empty, execute all procedures in M_x on v ;
 - ii. **else** execute default methods on v to get attribute values;
 - (b) restrict $d(a_1), \dots, d(a_k)$ with the new set of values;
 - (c) **if** any $d(a_i)$ becomes empty, **return failure**;
 2. **if** M_A is not empty, execute all procedures in M_A on a_1, \dots, a_k ;
 3. **else if** $d(a_1), \dots, d(a_k)$ are finite
 - (a) for each tuple $\langle u_1, \dots, u_k \rangle \in d(a_1) \times \dots \times d(a_k)$
 - i. execute the default method on $\langle u_1, \dots, u_k \rangle$ to get a x value;
 - (b) restrict $d(x)$ by the new set of values;
 - (c) **if** $d(x)$ becomes empty, **return failure**;
 4. add variables whose domains are changed A ;
 5. **return success**;
-

For example, given the variable representing the structure $\langle 8, \text{USA}, \emptyset \rangle$, *i.e.*, all 8-km maps of the US, we can trivially determine the domains for the *resolution* and *region* attributes, but we have no way of determining the domain of the *date* attribute. But given a variable representing $\langle \emptyset, \text{USA}, 2003154 \rangle$, we can determine (via a procedure call that translates into a database query), what maps of the US are available for that date. The API call returns a set of the actual data structures (in the case of TOPS, Java objects) representing each map. This set comprises a finite domain representation for the structure variable. We can then determine the resolution for each map by executing the appropriate procedure on the corresponding data structure, making that domain finite as well. A more formal description of structure constraint execution is given in Algorithm 3.

A constraint gets executed by the propagator when any variables in the constraint scope are added to the agenda, which usually means their domains are changed. Executing a structure constraint not only enforces the compatibility between the structure and its attributes; it also initializes the domains of variables, which may have unknown domains at the beginning, and eliminates inconsistent values from the domains.

3.3 Comparison to the hidden variable representation

The hidden variable representation is a binary representation of non-binary constraint problems [6,1]. A k -ary constraint represented as a relation can be translated into a set of k binary constraints by adding a hidden variable, with a domain containing the tuples in the original constraint. A hidden variable represents the original constraint, and each binary constraint (also called a hidden constraint) between the hidden variable and each regular variable is one-way functional constraint, in which each value (a tuple in the

relation) of the hidden variable is compatible with at most one value in the domain of the regular variable.

If an application problem involving structures doesn't have constraints over structures but does have constraints over structure attributes, the problem can be formalized as a non-binary CSP where attributes are variables and structures are constraints. Such a non-binary CSP can be transformed as a binary one via hidden variable transformation, where hidden variables are actually structures. For example, in a crossword puzzle problem, which is to find words from a given set of words that fit into the "word slots" on a $n \times m$ puzzle board with blanks, words may be viewed as structures and letters as attributes. There are only constraints on letters (i.e., on word slot intersections). This problem can be formalized as a non-binary CSP, where variables are the spaces on the puzzle board (excluding blanks) and constraints are the word slots, which can be represented using relations (each tuple is a word of the same length in the given dictionary). This CSP can be transformed into a binary CSP, where hidden variables are word slots.

If the problem has constraints over structures, representing structures as constraints may lead to 2nd-order constraints, which should be avoided. The structure representation we propose here indeed results in a non-binary CSP, but it is not suitable for hidden variable transformation, because structure constraints are usually not specified by relation tables.

However, we can have a binary structure representation: instead of a non-binary constraints on the structure variable and its attribute variable, we can have a set of binary constraints, each is posed on the structure variable and one of its attribute variables. This representation may be useful in some limited situation; for example, if a binary CSP solver is preferred.

Our reason for using k -ary structure constraints instead of a binary representation is that the latter is inadequate to represent the ESDP domain. In practice, the structures we need to represent correspond to real data structures, in our case, Java objects in the TOPS system, and the constraint procedures are function calls on those data structures made available by TOPS. While some of those function calls are a good fit for the hidden variable representation, namely, "getter methods," i.e., functions that take a structure as an argument and return one of its attributes, other function calls take multiple arguments and cannot be represented using binary constraints. By defining a general structure constraint in the constraint library, we are able to integrate the constraint reasoning system with the runtime software environment so that the operations provided by the environment can be called when the constraint is executed during constraint propagation. The constraint system is able to *query* the environment, to dynamically determine what objects exist and what attributes or properties those objects have.

3.4 Example - the model ingest problem

A core problem of Earth science data processing is "model ingest", i.e., preparing a number of data files representing Earth system variables (we will use the word *s*-variable to avoid confusion with constraint variables) and feeding them into a model. Typically, the model inputs are *spatial* data, such as satellite images, and other structured data types, as shown in Figure 1.

Most approaches to model ingest specialize on particular data sources, hard-coding choices such as resolution and projection. This is not required by the models themselves, which typically compute their results on a pixel-by-pixel basis, where the individual pixels can be regarded as point data. What *is* required by the models is that the inputs are consistent, with the same resolution, projection, etc., so that corresponding pixels all refer to the same location and time. Committing to particular data sources ensures that the inputs are mutually consistent, but at the cost of flexibility; changing, say, from 8km data to 1km data requires a major recoding effort.

Since the reason for using a planner is to have a system that can use the best data for a given purpose, subject to availability and resource constraints, such as storage and bandwidth, we adopt the opposite approach, making no commitment to a particular data source, resolution or projection, but relying on constraints to ensure that they are compatible. In addition to selecting input files that are compatible, we also have the option of *making* them compatible, for example by reprojection, which is represented as a planner action.

We consider a simplified version of this problem, where we are given a set of 20 georeferenced spatial data files. and the goal is to find (or produce) 4 data files to feed into the model. For simplicity, we assume that these files are specified by a flat list of 10 attributes such as *resolution*, *projection*, etc, and all of them are primitive types. These 4 files need to satisfy the following conditions:

- We need one file each for the s-variables FPAR, LAI, PRECIP and TEMP.
- the 4 files have to be the same *resolution*, *projection*, and *format*.
- the 4 files have to cover the same geographic region.
- the *date*, *cloudiness*, and *quality* of these 4 image files have to be in the specified range. In particular, FPAR and LAI don't change very rapidly, so any file within a week of the target date is acceptable, but *cloudiness* has to be low. Conversely, PRECIP and TEMP must be for the exact date of interest, but clouds are irrelevant.

There are two actions available: *reproject* which changes the projection of the input file, and *scale* which decreases the resolution.

This simplified problem is trivial for the IMAGEbot Planner, but for the purpose of being able to ignore other variables and constraints generated by the planner, let's assume that the planner constructs a grounded planning graph (that is, every action node is grounded) as in Figure 4. Note that the diagram only shows actions for one goal; others are the same due to symmetry of the 4 subgoals.

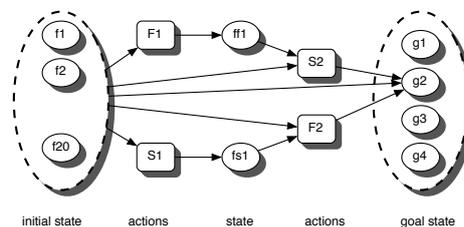


Fig. 4. A simplified planning graph

Ignoring the actions, the problem in hand is a typical constraint problem, where 4 files need to be found from the given 20 input files, satisfying those conditions. Because there are no constraints between files, only between attributes, we don't even need structure variables and structure constraints. The minimal CSP would have 4×10 variables, each for an attribute of the file, and the user-specified constraints listed above, such an AllSame constraint on the resolution attribute variables. The input file objects can be represented by 10-ary constraint containing 20 tuples.

With actions involved, we still need to find the 4 files, but the files can be taken directly from the input file set or can be produced by the chosen actions. Therefore, we have constraints specifying the relation of inputs and outputs of planner actions, and specifying the casual relation of subgoals and conditions, all involving file objects. For the graph in Figure 4, we have 16 actions; for each action, we have the following variables and constraints:

- a structure variable for its input image, and 10 variables for the input's attributes, and a boolean variable b representing whether the action is in the plan.
- a structure constraint for the structure variable and its attributes, and 10 conditional equality constraints, each with the conditional variable b and a pair of attributes, one from the input image and the other from the output of the action supporting it. For example, suppose the variable $scale$ is true iff the goal file g_4 is produced by the $scale$ action and the input of action $scale$ is $scaleIn$. Then we have the constraint $scale \Rightarrow .5 * scaleIn.res = g_4.res$, meaning that if the $scale$ action is chosen to produce g_4 , then the resolution of g_4 will be half the resolution of $scaleIn$. We also have the constraint $scale \Rightarrow scaleIn.format = g_4.format$, meaning that the format of g_4 will be the same as that of $scaleIn$.
- other boolean variables for logical expressions that represent planner subgoals and planner action conditions, which we will ignore them for now.

4 Solving the constraint problem

4.1 Reasoning about structures

Structure unification, as discussed in Section 2.2, and as illustrated above, creates a fairly large number of equality constraints over structures and structure attributes, which provides an opportunity for symbolically reasoning about structures.

When the planning graph is built, we create an additional *symbolic* constraint network containing only equality constraints, mainly on structures (note that a regular variable can be considered a structure of a primitive type). This network of equality constraints is propagated before creating the regular CSP. The propagation aims mainly at eliminating unnecessary variables and constraints. It also detects inconsistencies. The propagation and unification algorithms are given in Algorithm 4 and 5.

4.2 Propagation and search

Constraint propagation is an essential part of solving the constraint problem generated from a planning problem, not only because it eliminates inconsistent values, but also

Algorithm 4 Propagation by unification

Let \mathcal{N} be a network of equality constraints, A a set of structures called *agenda*, and let L and R be two structures

propagate(A)

1. **while** A is not empty, do
 - (a) C ← set of constraints containing structures in A ;
 - (b) **for each** $L = R \in C$
 - i. remove $L = R$ from C ;
 - ii. **if** *unify*(L, R, A, \mathcal{N}) returns failure, **return failure**;
 2. **return success**;
-

Algorithm 5 Unification

Let \mathcal{N} be a network of equality constraints, A a set of structures called *agenda*, and let L and R be two structures

unify(L, R, A, \mathcal{N})

1. **if** $L \equiv R$, remove $L = R$ from \mathcal{N} , **return success**;
 2. **if** L and R are not the same type, **return failure**;
 3. **if** L and R are constants and $L \neq R$ **return failure**;
 4. **if** R is a constant
 - (a) **if** L is a free variable, substitute R for L in \mathcal{N} ; remove $L = R$
 - (b) add all structures changed by substitution to A ;
 5. **if** L is a constant
 - (a) **if** R is a free variable, substitute L for R in \mathcal{N} ; remove $L = R$
 - (b) add all structures changed by substitution to A ;
 6. **for each** pair of attributes L_a and R_a , add $L_a = R_a$ to \mathcal{N} ;
 7. **return success**;
-

because the constraint problem in hand contains universally quantified variables with infinite domains, which cannot be enumerated by search. The propagation on regular variables (including structure variables, which are treated as regular variables) is straightforward: whenever the domain of a variable is changed, the constraints containing this variable are executed. If executing the constraint results in any variable domain changes, the constraints containing the changed variables will be executed. This process continues until there are no further changes, or a constraint execution fails; that is, the execution results in empty variable domains. Constraint execution failure implies that the constraint network is not consistent. The propagation enforces generalized arc-consistency [2,14].

We have implemented a heuristic planning search algorithm and a few constraint search algorithms, such as backtracking, backjumping and conflict-directed backjumping, all of which interleave search with propagation; that is, when the search algorithm assigns a value to a variable, the changes are propagated. The high-level planning search, guided by heuristic distance estimates extracted from the planning graph [3], selects planner subgoals to achieve, and planner actions to achieve the subgoals. The constraint search finds values for variables representing planner action parameters. This is necessary to make actions executable. During the search, propagation is performed whenever a value is assigned to a variable. The search is an iterative process involving possible backtracks; that is, if there are no valid parameters for a chosen action, the planner has to search for another plan; if it is impossible to extract a plan from the current planning graph, the planning graph has to be extended.

5 Conclusions

We have discussed a novel approach to constraint-based planning in domains that involve structured objects. The planner is implemented and used in the IMAGEbot planner-based agent, which has been applied to the TOPS ecological forecasting application. Although our focus in this paper was the Earth-science data processing (ESDP) domain, the planner is domain-independent, and the approach presented is broadly applicable. Structured objects, in the form of complex data structures, are ubiquitous in software domains, and may be useful in areas such as custom hardware configuration and manufacturing.

Contributions of the paper include the introduction of structure variables and structure constraints over those variables and a symbolic propagation algorithm that simplifies the constraint network.

There has been relatively little work relating to structured objects in either planning or constraint reasoning. One notable exception is the COLLAGE planner [15], which dealt with structured object in the KHOROS image processing domain and was also based on constraint reasoning. Another constraint-based planner for a similar domain is the Multi-mission Vicar Planner (MVP) [5]. Both of these are action-decomposition planners, which have less of a need to represent and reason about data transformations in the way that IMAGEbot does, and neither supports the kind a structure constraints discussed in this paper. [4] addresses workflow planning for computation grids, a sim-

ilar problem to ours, though their focus is on mapping pre-specified workflows onto a specific grid environment, whereas our focus is on generating the workflows.

References

1. F. Bacchus, X. Chen, P. van Beek, and T. Walsh. Binary vs. non-binary constraints. *Artificial Intelligence*, 140:1–37, 2002.
2. C. Bessiere and J. Ch. Arc-consistency for general constraint networks: Preliminary results. In *Proceedings of IJCAI-97*, pages 398–404, Nagoya, Japan, August 1997.
3. A. Blum and M. Furst. Fast planning through planning graph analysis. *AIJ*, 90(1–2):281–300, 1997.
4. J. Blythe, E. Deelman, Y. Gil, C. Kesselman, A. Agarwal, G. Mehta, and K. Vahi. The role of planning in grid computing. In *Proc. 13th Intl. Conf. on Automated Planning and Scheduling (ICAPS)*, 2003.
5. S. Chien, F. Fisher, E. Lo, H. Mortensen, and R. Greeley. Using artificial intelligence planning to automate science data analysis for large image database. In *Proc. 1997 Conference on Knowledge Discovery and Data Mining*, August 1997.
6. R. Dechter. On the expressiveness of networks with hidden variables. In *Proceedings of AAAI-90*, pages 556–562, 1990.
7. M. Do and S. Kambhampati. Planning as constraint satisfaction: Solving the planning graph by compiling it into CSP. *Artificial Intelligence*, 132:151–182, 2001.
8. J. Frank, A. Jónsson, and P. Morris. On reformulating planning as dynamic constraint satisfaction. In *Symposium on Abstraction, Reformulation and Approximation*, Texas, 2000.
9. K. Golden. DPADL: An action language for data processing domains. In *Proceedings of the 3rd NASA Intl. Planning and Scheduling workshop*, pages 28–33, 2002. to appear.
10. K. Golden and J. Frank. Universal quantification in a constraint-based planner. In *AIPS02*, 2002.
11. K. Golden and W. Pang. Constraint reasoning over strings. In *Proceedings of the 9th International Conference on the Principles and Practices of Constraint Programming*, 2003.
12. Keith Golden. An domain description language data processing. In *ICAPS 2003 Workshop on the Future of PDDL*, 2003.
13. A. Jónsson. *Procedural Reasoning in Constraint Satisfaction*. PhD thesis, Stanford University, 1996.
14. G. Katsirelos and F. Bacchus. GAC on conjunctions of constraints. In *CP-2001*, 2001.
15. A. Lansky. Localized planning with action-based constraints. *Artificial Intelligence*, 98(1–2):49–136, 1998.
16. A. Lopez and F. Bacchus. Generalizing graphplan by formulating planning as a CSP. In *Proceedings of IJCAI-2003*, 2003.
17. R. Nemani. Data processing is 80 percent of the work. Personal Communication, 2003.
18. R. Nemani, P. Votava, J. Roads, M. White, P. Thornton, and J. Coughlan. Terrestrial observation and prediction system: Integration of satellite and surface weather observations with ecosystem models. In *Proceedings of the 2002 International Geoscience and Remote Sensing Symposium (IGARSS)*, 2002.
19. J.S. Penberthy and D. Weld. UCPOP: A sound, complete, partial order planner for ADL. In *Proc. 3rd Int. Conf. Principles of Knowledge Representation and Reasoning*, pages 103–114, October 1992.
20. D. Smith, J. Frank, and A. Jónsson. Bridging the gap between planning and scheduling. *Knowledge Engineering Review*, 15(1):61–94, 2000.
21. P. van Beek and X. Chen. CPlan: A constraint programming approach to planning. In *Proceedings of AAAI-99*, 1999.