A Hybrid Constraint Representation and Reasoning Framework

Keith Golden[1] and Wanlin Pang[2]

[1] Computational Science Division, NASA Ames Research Center, Moffett Field, CA 94035
[2] QSS Group Inc., NASA Ames Research Center, Moffett Field, CA 94035

Abstract. This paper introduces JNET, a novel constraint representation and reasoning framework that supports procedural constraints and *constraint attachments*, providing a flexible way of integrating the constraint reasoner with a runtime software environment. Attachments in JNET are constraints over arbitrary Java objects, which are defined using Java code, at runtime, with no changes to the JNET source code.

1 Introduction

Constraint-based reasoning has been shown to be useful in representing and reasoning about such diverse problems as the *graph-coloring problem* [10], the *satisfiability problem* [5], the *scene labeling problem* [19], and the *resource allocation problem* [20]. In theory, the problem in hand is formalized as a constraint satisfaction problem (CSP) and is solved by using CSP algorithms such as backtracking. In practice, a few constraint systems [1,2,18] have been developed and used as tools for implementing industrial applications. A typical constraint system consists of a search engine and a constraint library containing domain-independent constraints, such as *all different*, *sum*, *cardinality*, etc. A well-recognized limitation of applying such constraint reasoning tools is that many real-world applications often involve constraints that may not be modeled with built-in constraints in the constraint library. Those domain-specific constraints have to be implemented and added to the constraint library, and in certain circumstances, the underlying constraint search engine has to be tailored to deal with these specific constraints. Even if a constraint reasoning tool allows such extension and modification, it is a great burden for the user of a constraint-reasoning tool to extend the tool itself.

In some other real-world applications, for example, the application of constraint-based planning to processing earth-observing satellite data [6,8], where the constraints involved are arbitrarily complex and dynamic, extending the constraint library may not be feasible. We are applying constraint-based planning to the Earth-science data-processing domain. This is a domain in which constraints may arise among complex objects, such as satellite images and weather forecast data. Because the world is large and dynamic, it is impossible to enumerate in advance all possible objects, such as satellite images, much less provide an extensional representation of the constraints among them. Moreover, many of the constraints we would like to use are very complex, but are implemented as executable code in a software environment. Reimplementing them in our constraint reasoning system would not only be difficult, but would also violate

the principle that information should exist in only one place. For example, objects in the Earth Science Data Processing domain includes "tiles" from Earth-orbiting satellites. A tile is a rectangular satellite image in some specified projection covering some definite region of the Earth. A tile has a number of attributes, including the projection, the instrument used to capture the image, the time and location where the image was captured, the pathname where the image is stored and a unique identifier that can be used to reference the tile. The pathname and unique identifier both depend on other attributes of the tile, but the specific rules used to generate these are complex and subject to change over time. For example, the pathname of the tiles depends on what disk is used to store them. Encoding these rules directly in the constraint system would lead to "bit rot," i.e., causing the planner to stop working whenever details of the rules change. Instead, we should simply invoke the operations provided by the software environment.

We would like to integrate the constraint reasoning system with the runtime software environment so that the operations provided by the environment can be used as constraints. We would like the constraint network to "query" the environment, to dynamically determine what objects exist and what attributes or properties those objects have. Doing so requires being able to define types in the constraint network that correspond to entities within the runtime environment and to define constraints in terms of operations supported by the runtime environment.

The procedural contraint reasoning framework introduced in [11] and extended in [12] comes close to providing the capabilities we need. Constraints can be defined in terms of procedures, which can be implemented as arbitrary (C++) code. Such code could be used to make calls to the external software environment. However, there are a few disadvantages to this framework.

- Variables cannot have values that are objects in the environment. Only variables from a predefined set of basic types provided by the constraint network can be defined. This is very limiting if we want to define constraints that relate objects in the environment.
- Procedural constraints are implemented as classes in the constraint network package, and must be defined according to the data structures for variables, values and domains that are used by the constraint network. This requires anyone who writes constraints to be fairly knowledgeable about the inner workings of the constraint network. It also requires access to the source code of the constraint network and recompilation of the source code when constraints are updated.
- The amount of code needed to write a procedural constraint is large compared to the roughly one line of code needed to invoke a typical operation in a software environment.

We have implemented a hybrid constraint reasoning system, called JNET, for Java constraint NETwork, which builds upon the constraint network described in [12] and addresses the above concerns. JNET provides

- Arbitrary, complex types, defined at runtime, corresponding to Java classes. Any object in the Java runtime environment can appear as a value in a variable domain.
- Arbitrary, complex constraints, defined at runtime using *constraint attachments*, constraints specification in terms of functional Java methods, which are concise

and simple to specify constraints without any knowledge of the workings of JNET. However, they interact well with the JNET propagation algorithm.
- A library of common constraints.
- Interaction with the runtime environment: many of the constraints we would like to represent, such as constraints involving files, images, etc., involve arbitrary objects internal to the runtime environment, and the constraints themselves may be impossible to define except by reference to operations provided by the environment.
- Open-world scenarios: given the large number of files available in data-processing environments, it is infeasible to explicitly enumerate in advance all of the objects in the universe and the relations among them. Instead, we query the environment for those objects relevant to a particular planning problem. We can do this naturally and flexibly using constraint reasoning, by representing these "queries" as constraints. Open worlds can be divided into two cases: unknown variables and unknown domains. Unknown variables can dealt with by adding variables within a dynamic CSP. Unknown values can be dealt with beginning with open domains for some variables and allowing sensors to serve as constraints, restricting the domains as information is acquired.
- Dynamic CSPs: The framework can handle the addition and deletion of variables, values and constraints

The remainder of the paper is organized as follows: In Section 2, we look at an example of a planning problem in the data processing domain that motivates the need for a hybrid constraint-reasoning framework. In Section 3, we discuss the constraint framework. In Section 4, we then discuss in more detail the constraints specific to the data-processing domain and how they are represented in our framework. In Section 5, we discuss the implemented constraint system. In Section 6, we conclude by summarizing our contribution and discussing the limitations and future work.

2 Planning as Constraint Reasoning

Earth-science data processing 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 these 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 planing 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.

Constraints arise naturally in this planning problem. 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. For example, scal-

ing an image creates a new image whose dimensions are some multiple of the dimensions of the original. In the course of planning, additional constraints arise specifying how parameters of an action depend on the parameters of other action in the plan.

We are working with Earth scientists to provide planner-based automation to an ecosystem forecasting system called the Terrestrial Observation and Prediction System, or TOPS [17] (http://www.forestry.umt.edu/ntsg/Projects/TOPS/). We have developed a planner-based softbot (**soft**ware ro**bot**), called IMAGEbot [8], to generate and execute data-flow programs (plans) in response to data requests. The data processing operations supported by IMAGEbot include image processing, text processing, managing file archives and running scientific models.

The architecture of IMAGEbot is described in Figure 1. Planning domains are loaded by the parser and passed to the planner. The planning domains include definitions of actions, as well as complex types, functions and relations, which can, in turn, be defined in terms of named constraints from a constraint library or constraint attachments, specified using Java code. Given these definitions, goals from a user and an "initial state" specification from a database or a file loaded by the parser, the planner converts the planning problem into a dynamic CSP (DCSP), which it gives to JNET to solve. The planner controls the high-level search, guided by heuristics derived from a Graphplan-style [4] reachability analysis. During planning, information from the environment comes to the planner via JNET in the form of variables whose values are determined by constraint attachments. When new objects are discovered that correspond to complex types, the planner may introduce additional variables and constraints to the DCSP to reason about these objects in JNET. The planner may also add variables and constraints to the DCSP in response to search failure or execution failure. A solution to the DCSP corresponds to a solution to the planning problem. The planner sends executable plans or sub-plans to the executive, which may return information that is fed back into (re)planning and constraint reasoning.

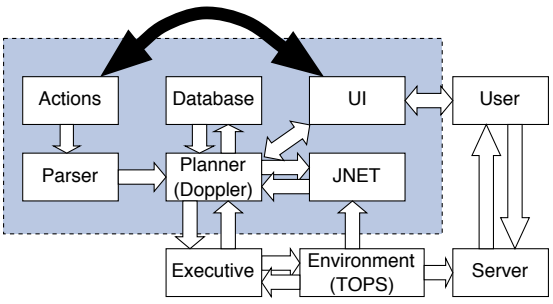

Fig. 1. The agent architecture

3 Procedural Reasoning Framework

The planning problem discussed above is reformulated in the IMAGEbot Planner as a constraint satisfaction problem (CSP) that is handled by the constraint reasoning subsystem called JNET. In this section, we present the basic idea behind JNET, namely, the *constraint procedures* and *constraint attachments*.

3.1 Constraint SatisfactionProblems

A **Constraint Satisfaction Problem (CSP)** is a representation and reasoning framework consisting of variables, domains, and constraints. Formally, it can be defined as a triple $< X, D, C >$ where $X = \{x_1, x_2, \ldots, x_n\}$ is a finite set of variables, $D = \{d(x_1), d(x_2), \ldots, d(x_n)\}$ is a set of domains containing values the variables may take, and $C = \{C_1, C_2, \ldots, C_m\}$ is a set of constraints. Each constraint C_i is defined as a relation R on a subset of variables $V = \{x_i, x_j, \ldots, x_k\}$, called the constraint scope. R may be represented extensionally as a subset of Cartesian product $d(x_i) \times d(x_j) \times \ldots \times d(x_k)$. A constraint $C_i = (V_i, R_i)$ limits the values the variables in V can take simultaneously to those assignments that satisfy R. The central reasoning task (or the task of solving a CSP) is to find one or more solutions.

Many algorithms and systems have been developed for solving constraint problems, ranging form simple backtracking search algorithms to sophisticated hybrid methods. However, constraints involved in real-world applications, such as the data-processing domain discussed in previous sections, represent new challenges as to how to represent these constraints and to find solutions to the constraint problems. In the following, we present our constraint representation and reasoning framework.

3.2 Constraint Procedures

The idea of procedural reasoning in constraint satisfaction [11] is to augment a general constraint search engine with specific procedural methods that can quickly solve certain types of subproblems and prune a search space that contains no potential solutions. In certain sense, similar techniques have been widely used in solving binary CSPs; that is, enforcing arc-consistency while searching for solutions by backtracking [16]. A binary constraint is arc-consistent if for each value of one constrained variable there exists a value of the other constrained variable such that the value pair satisfies the constraint. To enforce arc-consistency, we eliminate those values from a variable domain for which there is no corresponding value in the other variable domain satisfying the constraint. Such values are usually called inconsistent values. There have been many algorithms published in the literature [15,13,14,3] for enforcing arc-consistency, but the question of how to detect and then remove inconsistent values has largely been ignored, because it seems to be a trivial implementation issue when dealing with binary constraints, which can be uniformly represented as a 0-1 matrix (assuming finite domains). However, when it comes to non-binary constraints, enforcing a local consistency relative to a constraint is not obviously a trivial task. Instead, constraints in different application domains are represented and enforced in different ways. In addition, many constraint satisfaction problems contain simple functional relations (e.g.

arithmetic equations) and simple subproblems (e.g. linear equations with unknowns) that can be solved quickly by using existing algorithms. The question becomes 1) how to uniformly represent constraints that arise in different applications; 2) how to take advantage of such algorithms in order to significantly improve search efficiency. The procedure reasoning framework, which has been formalized in the context of constraint satisfaction [11], addresses this question.

In general, the notion of a constraint procedure encompasses a wide range of constraint reasoning techniques, from simple propagation to complete search methods:

A constraint procedure p is a function that maps a CSP $\mathbb{P} = (X, D, C)$ to another CSP $\mathbb{P}' = (X, D', C')$ such that: 1) $d(x_i) \subseteq d'(x_i)$ for each $x_i \in X$, $d(x_i) \in D$, $d'(x_i) \in D'$; 2) for each constraint $C_h = (V_h, R_h) \in C$ there exists a constraint $C'_h = (V_h, R'_h) \in C'$, such that C_h and C'_h have the same scope and $R'_h \subseteq R_h$.

A constraint procedure p is correct if the set of solutions to $\mathbb{P}'$ is the same set of solutions to $\mathbb{P}$. This definition permits a constraint procedure to eliminate values from variable domains, restrict existing constraints, and add new constraints. The correctness criterion ensures that these operations defined in a procedure applied to a CSP only transforms the CSP to an equivalent one; that is, the set of solutions to the CSP will not be affected.

The concept of constraint procedures provides a uniform and efficient method to represent and reason with constraints. In terms of representation, constraint procedures can be used to specify any kind of constraints over any kind of variables. In fact, by the CSP definition in Section, a constraint on a subset of variables can be seen as a function that maps a universal relation on the variable subset into a restricted relation on the same variables defined by the constraint. From the reasoning point of view, constraint procedures can be applied for the purpose of both maintaining consistency and searching for a solution. Given a constraint $C_i = (X_i, R_i)$ where $X_i = \{x_{i_1}, x_{i_2}, \ldots, x_{i_m}\}$ and $R_i \in d(x_{i_1}) \times d(x_{i_2}) \times \ldots \times d(x_{i_m})$, executing the procedure representing the constraint eliminates those values from variable domains that won't be in any tuple in R_i; in particular, for any tuple assigned to the variable subset, enforcing the constraint ensures it is a consistent partial assignment. When interleaving the execution of constraint procedures with a search algorithm such as standard backtracking, it dramatically increases efficiency of solving the constraint problem. This is so because many variables will be assigned a value by the constraint procedures instead of by the search engine.

3.3 Constraint Attachments

A constraint essentially specifies a relationship among constrained variables that should be maintained, for example, $x + y = z$ is a constraint describing an equality relation that holds among three numeric variables x, y, and z. As discussed previously, a procedural representation of this equality constraint is a constraint procedure maintaining the equality relation; that is, whenever the domain of x, y, or z changes, the constraint procedure will be executed to detect and eliminate inconsistent values from the domains of other variables. In particular, when the domains of these variables become singleton, that is, the variables have been assigned a single value, the constraint procedure ensures that the equality relation holds. A constraint attachment is an alternative formalism for representing a underlying constraint. It can be considered a special case of a constraint

procedure in that the attachment consists of a set of functional methods, which collectively define the relation. Each method takes a list of arguments as input variables and it returns the calculated result for its output variable, such that the underlying constraint is still satisfied. For example, the constraint $x+y=z$ would, in general, include three methods: $z \leftarrow x+y$, $x \leftarrow z-y$, and $y \leftarrow z-x$. The method $z \leftarrow x+y$ calculates z's domain based on the domains of the given variables x and y, and it is usually invoked when either or both domains of x and y changes.

The idea of constraint attachment can be traced back to *procedure attachment* in [9], and it has been noted in [11] that such an approach in constraint satisfaction has certain shortcomings in terms of reusability, global algorithm implementation, and integration with search engines. The JNET framework addresses these concerns. A constraint attachment is a special constraint procedure in that the procedure consists of a set of functional methods, therefore, it works with any search engines for which a constraint procedure works. Furthermore, we don't need constraint attachments for implementing any algorithms or constraints that can be implemented with general constraint procedures. In other words, global algorithms and any reusable constraints can be implemented as constraint procedures.

As we will discuss in Section 5, constraint procedures are usually implemented to approximately enforce generalized arc consistency for more efficient execution. Constraint attachment has certain advantageous over general constraint procedures in terms of flexibility. Given a constraint attachment, not only can a set of functional methods be selectively implemented, but also the implemented methods can be selectively executed by a constraint propagator or a constraint solver without any tailoring of the propagator and the solver to the specific constraints. This selective execution can exploit the knowledge of what variable domain a given method will affect and what variable domains that method depends on. Most importantly, by combining constraint attachments and constraint procedures, we have a hybrid constraint representation and reasoning mechanism that significantly improves applicability of constraint systems and also allows constraint systems to interact with a dynamic runtime environment.

Since the focus of this paper is constraint attachments in JNET, we now discuss how attachments are defined in domain descriptions, how they are used by JNET, and how execution of the attachments is used to communicate with the software environment.

4 Domain Specific Constraints

Domain descriptions are specified in a language called the Data Processing Action Description Language (DPADL) [7], which allows the description of planning domains that involve data processing operations as well as the constraints appearing in those domains. We limit the discussion of DPADL syntax to that needed to show how constraints are defined.

We provide two alternative ways of specifying the definition of a constraint; it may be selected from the built-in JNET constraint library if such a constraint is defined, or it may be defined in terms of arbitrary Java code embedded in the type, attribute and function declarations if such a constraint does not exist in the constraint library. The

constraint network supports constraints over all primitive types as well as Java objects. It can also handle constraints involving universal quantification, as discussed in [6].

4.1 Types

DPADL is an object-oriented language modeled on Java, so it supports complex types, which may (but need not) correspond to Java classes. When types do correspond to Java classes, attachments are used to define constraints on those objects. Complex types may have multiple attributes (i.e., fields or members), which may themselves be instances of complex types. We will not discuss the details of this representation because, at the level of constraint reasoning, it doesn't really matter. We can describe a complex type using a set of relations that relate an instance of the type to each of its attributes, so ultimately, all we have is variables, values and relations. What does matter is that the types of variables may be Java classes, the values may be Java objects, and the relations may be defined using Java methods. In addition to complex types, we can also have primitive types, corresponding to integers, floating point numbers, strings and booleans, and we may define subtypes any type, which may, but need not, be represented extensionally. For example, we may define the type `imageFormat` as the set {"JPG", "GIF", "PNG", "TIFF", "HDF", "XCF"}.

Constraints may be defined for any type. Constraints associated with primitive types are unary, but constraints associate with complex types can refer to attributes of the type as well as the instance of the type itself. Constraints associated with a type will be instantiated for each variable of that type in the constraint network.

For example, suppose we want to specify a filename as a subtype of string. JNET supports extensive capabilities for representing and reasoning about string constraints using a domain representation based on regular languages. The most fundamental string constraint is the unary matches constraint, which specifies that the string matches a given (constant) regular expression. Thus, we can say that all filenames satisfy the constraint

```
matches(this, "~[/]")
```

which specifies that filenames must contain at least one character, and they cannot contain the character '`/`'. In Unix, this is, in fact, the only practical limitation on filenames. The keyword **`this`** is a special variable that refers to an instance of the type being defined, so if the above constraint appears in the type definition for `filename`, then all variables of type `filename` will have that constraint.

In addition to constraints from the library, we can define constraints using attachments, as we discuss below.

4.2 Attributes, Functions and Relations

DPADL is a functional language, so relations and attributes are ultimately represented as functions. This is not an important distinction, but since we refer to relations in a different sense in our definition of constraints, we will refer to attributes, functions and relations declared in domain descriptions collectively as *functions*. Functions lead

a double life in DPADL. Because not all aspects of planning problems require constraints, we can declare functions that do not correspond to constraints. Subgoals containing these functions are handled by the planner, via unification and goal regression. They may ultimately appear as *variables* in the constraint network generated by the planner, but never as constraints. On the other hand, some functions, such as arithmetic expressions, clearly are best represented as constraints. We do this by specifying one or more constraints that *define* the function. If any function has a constraint definition, the planner does not attempt to deal with it, but adds the corresponding constraints to JNET. Like constraints associated with types, constraints associated with functions can be specified using either the constraint library or attachments

4.3 Constraint Attachments in Data Processing Domain

To specify a constraint attachment that can be used by JNET, we must specify the set of variables involved in the constraint and the method (the actual code) used to implement the constraint. Formally, an attachment is a pair $< P, m >$, where P is a *signature* and m is a *method*. Conceptually, P specifies the arguments and return values of the method m as a list, $\{a_0, a_1, \ldots a_n\}$, where the first argument, a_0 designates the variable that will be assigned the return value of m and $a_1, \ldots, a_n$ designate the variables that will provide the arguments to m. The arguments a_i are not just variables, however. If we were only interested in implementing attachments that took singletons as arguments and returned singletons as results, then all we would need for P would be a list of variables. Instead, we allow the domain modeler to specify that an argument represents an entire domain, which may be in the form of a finite set or an interval. Thus, in addition to the variable, it is also necessary to specify what form the domain should take: a singleton, set or interval. Each argument a_i, then, is a pair $< t_i, x_i >$, where x_i is a variable from the constraint network and t_i specifies the form that the the domain of x_i should take. For method arguments (a_i, where $i > 0$), $t_i \in \{1, \Im, \mathcal{S}\}$, where 1 is used to denote a singleton, $\Im$ denotes an interval, and $\mathcal{S}$ denotes a finite set. The method m will only be applicable if each of the domains $d(x_i)$ can be converted to the representation t_i required by m. The possible values for t_0 are slightly different. Methods can return single values or sets, but not intervals; instead, two methods are specified, one to compute the lower bound of an interval and the other to compute the upper bound. Thus, $t_0 \in \{1, \lfloor \Im \rfloor, \lceil \Im \rceil, \mathcal{S}\}$, where $\lfloor \Im \rfloor$ and $\lceil \Im \rceil$denote the lower and upper bound of an interval, respectively.

Syntax Constraint attachments are specified in domain descriptions using a concise syntax that we will explain by example. A formal specification of DPADL syntax can be found in [7]. For example, the TOPS environment provides a class `tops.modis.Tile` to represent tiles from the MODIS instrument aboard the Terra and Aqua satellites. This class contains various methods, such as getUID, to provide information about tiles. We can specify attachments that define the `uniqueId` of a tile by reference to the methods provided by the environment that relate to unique identifiers:

```
value(this) = $this.getUID()$;
this(value) = $Tile.findTile(value)$;
```

There is a one-to-one mapping between tiles and their unique identifiers. Given a tile, we should be able to obtain its unique identifier, and given a unique identifier, we should be able to obtain the corresponding tile. The embedded Java code provides instructions for performing these mappings. The `uniqueId` attribute of a `Tile` can be determined by calling the `getUID` method on the `Tile`, and a `Tile` object corresponding to a given `uniqueId` can be determined by calling the method `findTile`, with the `uniqueId` as an argument. Each line above is an attachment. The text preceding the "=" is the signature P, written in the form $x_0(x_1,\ldots,x_n)$, and the following code, delimited by "$...$," is the method m. As discussed above, the variable **`this`** refers to an object of the type being defined. In this case, the definition of the `uniqueId` attribute appears in the definition of the `Tile` type, so **`this`** is an object of type `Tile`. The keyword **`value`** is a variable that identifies the return value of the function being defined, in this case `uniqueId`. Thus, the signature **`value`**`(`**`this`**`)` means that the method accepts an argument of type `Tile` and returns the value of the `uniqueId` attribute of that `Tile`. Conversely, **`this`**`(`**`value`**`)` means that the method expects a String representing the `uniqueId` and returns the corresponding `Tile`.

The two attachments used to define the constraint for `uniqueId` can be written as:

1. $<\{<x_{uid},1>, <x_{tile},1>\}, m_1>$
2. $<\{<x_{tile},1>, <x_{uid},1>\}, m_2>$

where the methods m_1 and m_2 are the Java methods generated from the attachment definition; m_1 is a method that takes a `Tile` and returns a `String`, and $m_2$is a method that takes a `String` and returns a `Tile`. The value 1 for t_1 means that a value for x_{uid} can only be obtained if x_{tile} is singleton, and vice versa. It is also possible to define constraints that work for non-singleton domains, by indicating that an argument or return value represents an interval $\Im$ (delimited by `[]` in the DPADL code) or a finite set $\mathcal{S}$ (delimited by `{}`). For example, one attribute of a `Tile` is that it `covers` a given longitude, latitude. Given a particular longitude and latitude, the constraint solver can invoke a method to find a single tile that covers it, but it can do even better. Given a rectangular region, represented by intervals of longitude and latitude, it can invoke a method to find a set of tiles covering that region.

```
/**
 * true if this tile covers the specified lon/lat.
 */
boolean covers(float lon, float lat) {
  constraint {
    ...
    // returns the set of tiles covering given range.
    {this}([lon], [lat], d=day, y=year, p=product, value)
      = {$ if(value)
             return tm.getTiles(lon.max, lat.min, lon.min,

                                lat.max, d, y, p);
           else return null; $};
```

```
    }
}
```

In this example, the signature is more complicated. The use of {} around **this** indicates that the return value of the Java code is a set — specifically, a set of tiles, since the variable **this** indicates an instance of the type `Tile`. The first two arguments, lon and lat, are surrounded by `[...]`, indicating that the variable domains should be intervals. The next three arguments, `d`, `y`, and `p` are defined as being equal to the `Tile` attributes `day`, `year` and `product`, not shown in this example. Finally, **value** is the boolean value of the `covers` relation, true if and only if the tile covers the specified lon/lat.

The Java code is also more complex. Unlike the previous example, it has a conditional and an explicit return call. If **value** is true, then it returns the result of the method getTiles. Since lon and lat are intervals, we refer to their maximum and minimum values to specify the bounding box of interest. If **value** is false, it returns **null**, meaning the set of tiles could not be determined, since there is no method for returning the tiles outside of a bounding box.

In this example, $P = \{<x_{tile}, \mathcal{S}>, <x_{lon}, \Im>, <x_{lat}, \Im>, <x_d, 1>, <x_y, 1>, <x_p, 1>, <x_{covers}, 1>\}$ and m is a method that returns a Collection of Tile objects.

We can also specify attachments that calculate interval domains. We do this by specifying two attachments, one for the minimum value of the interval and one for the maximum value. For example, the type Date is represented as the number of milliseconds since an epoch, as specified by the class java.util.Date. If the domain of a Date variable is represented as an interval, then we can compute an interval representation of the year by determining the year of the minimum date and the year of the maximum date.

```
int year {
  constraint {
    [value]([this]) :=
      [$ cal.setTime(new Date(this.min));
         return cal.get(cal.YEAR); $,
       $ cal.setTime(new Date(this.max));
         return cal.get(cal.YEAR); $];
  }
}
```

Here, there are two attachments specifying the domain of the year attribute of Date

1. $< P_1, m_1 >$,where $P_1 = \{<x_{year}, \lfloor \Im \rfloor>, <x_{date}, \Im>\}$
2. $< P_2, m_2 >$, where $P_2 = \{<x_{year}, \lceil \Im \rceil>, <x_{date}, \Im>\}$

Implementation Details The parser generates attachments from their DPADL specification by generating Java methods from the in-lined code (delimited by $...$) that appears in attachment definitions, together with the signature, from which the method's parameter list and return type can be determined. The parser then compiles and loads the generated Java code, and uses reflection to obtain references to the newly defined

methods from their names. This last part points to an advantage of Java over, say, C or C++, which does not support reflection. Implementing run-time attachments in C would not be impossible, but it would be considerably more difficult. Lisp, and a few other high-level interpreted languages, would also be a suitable language for implementing attachments. For our purposes, Java has the unique advantage that the runtime environment we want to interface with is written in Java.

This approach enables constraint attachments to be defined at runtime, without requiring modification to, or even access to, the source code of the constraint network. In contrast, constraint procedures [11] are implemented as classes in the constraint network package; adding a new procedural constraint involves modifying and recompiling the constraint network source code. Although the values passed to the method *m* correspond to variable domains, the code that implements *m* is entirely independent of the implementation of the constraint network and the representation of variable domains.

1. If the specified domain type is 1, the argument passed to the method is simply the one value in the domain, which is either a primitive (int, boolean, etc.) or a Java object. The declared type of the corresponding parameter is simply that type.
2. If the specified domain type is $\mathcal{S}$, then the argument is an instance of java.util.Collection, where the members are either wrappers for the primitive types (Integer, Boolean, etc.) or Java objects corresponding to the declared object type.
3. If the specified domain type is $\mathfrak{I}$, the argument is an object that has two attributes, min and max, of type int or float, depending on the declared type of the variable. The user need not know anything else about the object.

Similarly, the return value for singletons is just the singleton value of the variable, and the return value for sets is a Collection. For intervals, two attachments are provided, as in the example of Date above: one to return the minimum value and one to return the maximum value, and these return value are either ints or floats.

For example, the methods m_1 and m_2 in the attachments for `uniqueId` are simply:

1. `String` m_1(Tile x)`{`**`return`** x`.getUID();}`
2. `Tile` m_2(String x)`{`**`return`** `Tile.findTile(`x`);}`

4.4 Requirements

Because attachments are used to define constraints, the attachments associated with a given type, attribute or function should collectively define the corresponding relation. Thus, there are certain requirements that the code is assumed to meet:

1. The code may not do anything other than calculate the domain of a variable and return it. That is, it may not have any side-effects.
2. If the code is called multiple times with the same arguments (which represent domains a subset of the of variables), it will always return the same value (which represents the calculated domain of another variable).
3. If the domains corresponding to one or more of the arguments is reduced, then the calculated domain will be a subset of the original domain.

If these requirements are not met, then the results are undefined.

Note that requirement 1 precludes the possibility of implementing stronger consistency enforcement than arc consistency, since there is no way to add a derived constraint. That is really a consequence of our decoupling the specification of constraint attachments from the implementation of the constraint network. Since JNET also supports procedural constraints, it is still possible to implement higher-order consistency, just not using attachments. Otherwise, these requirements are not very strong. We do not require that the definition of the constraint provide access to the full extension of the relation.[3] For example, we could have a binary constraint defined by only one attachment, and thus constraining only one variable. If the other variable is specified, then the value(s) for the constrained variable will be determined, but not vice versa.

5 Constraint System Implementation

We ported the procedural reasoning framework of [12], to Java, and extended it to support string domains, quantified constraints [6], and other features, including constraint attachments. In the following, we briefly describe resulting implemented system, called Java Constraint Network (JNET). JNET is a component in the IMAGEbot planner, but it can also be used a stand-alone constraint system with capacities of of solving a variety of constraint problems.

JNET contains classes for variables, domains, and constraints. Each variable is associated with a domain. A variable domain can be finite or infinite, in which case it is represented as an an interval (for numeric types), regular expression (for string types), or symbolic set (for object types). The constraints are implemented as procedures or constraint attachments. A constraint consists of a set of variables (the scope) and a procedure that enforces the underlying constraints on the variables. Enforcing a constraint eliminates inconsistent values from the domains of variables in the scope; that is, when the procedure is executed, it examines current domains and eliminates any values that are not consistent with values remained in other domains.

In the current version, JNET provides a fairly rich set of variable types and constraint library. Variables can be boolean, numeric (integers and floating point), string, and any Java objects. The implemented constraint library contains about thirty application-independent constraints, such as equality, less-than, maximum and minimum, cardinality, regular expression match and string concatenation. Since it is expensive (in the case of large finite domains) and even impossible (in the case of infinite domains) to completely enforce the underlying constraint in the procedure, many of the procedures only approximately enforce the constraint, based on an idea of maintaining generalized arc-consistency. Any procedural constraint fully enforces the constraint whenever each domain in the scopes has been reduced to a singleton. However, many constraints implemented and included in the constraint library can do much more to eliminate invalid values from variable domains without eliminating potential solutions.

[3] Except in the trivial sense that any particular assignment of values to the variables will either be consistent or it will not; if the assignment is inconsistent, then some attachment will return a domain for a variable that does not contain the assigned value for that variable.

Constraint attachment is implemented as a special procedure calling attached functional methods. This procedure is, in fact, a procedural constraint, used to represent all possible constraint attachments. When a set of attachments defining a new constraint is encountered, a new instance of the `attach` constraint is created, containing a list of all attachments associated with the constraint. Each attachment is of the form $< P, m >$, where P is the signature and m is the Java method as defined previously. Although, in our planner, m is generated by the IMAGEbot parser from the planning domain specification, it could be defined in any way, such as being implemented directly by the user with Java.

The search engine of the constraint system contains several search algorithms including depth-first search, backjumping and conflict-directed backjumping. A simple depth-first search is outlined as follows:

`DFS(searchableVars)`

1. **if** `searchableVars is empty,` **return success**;
2. **select** $x_i \in$`searchableVars;`
3. **for each** `value` $v \in d(x_i)$
   (a) **assign** $x_i = v$;
   (b) **if** `(propagate({x_i})` **and** `DFS(searchableVars -` x_i`))` **return success**;
4. **return failure**;

The set `searchableVars` contains all unassigned variables with finite domains containing more than one value. If a variable domain becomes a singleton during propagation, it is considered to have a value assignment. Therefore, propagation, together with procedures in each constraints, plays important role in the solution search process.

Propagation, as performed by the procedure `propagate()`, is a process of continuously executing constraint procedures as long as a variable domain changes. If propagation results in an empty variable domain, `propagate()` returns `failure`. Otherwise, it returns `success`. Given a set of variables V affected by either search or propagation, a simple propagation algorithm works as follows:

`propagate(affectedVars)` **while** `affectedVars is not empty` **do**

1. $x \leftarrow$ `a variable removed from affectedVars;`
2. $C_x \leftarrow$ `a set of constraints containing` x;
3. **for each** `constraint` $c \in C_x$
   (a) **if** `execute(`c`) success, add affected variables to affectedVars`
   (b) **else return failure;**
4. **return success**;

The correctness of the search algorithm outlined above is proved in [6], though its completeness is only ensured for CSPs with finite set solutions.

The constraint execution procedure for constraint attachments is as follows:

`execute(`$< P, m >$`)`

1. **for each** $< x_i, t_i > \in P$ where $i > 0$
   (a) **if** ($d(x_i)$ is not representable as the domain type designated by t_i) **return**
   (b) **else let** d_i be the singleton, set or interval representation of $d(x_i)$, where $t_i = 1, \mathcal{S}, \Im$, respectively.
2. **let** $v_0 \leftarrow$ **invoke** $m(d_1, \ldots, d_n)$
3. **assign** $d(x_0) \leftarrow d(x_0) \cap \begin{cases} \{v_0\} & \text{if } t_0 = 1 \\ v_0 & \text{if } t_0 = \mathcal{S} \\ [v_0, \infty) & \text{if } t_0 = \lfloor \Im \rfloor \\ (-\infty, v_0] & \text{if } t_0 = \lceil \Im \rceil \end{cases}$

Note that in 1(a), there is no requirement that the internal representation take the form of a singleton, interval or set, merely that it is possible to represent the domain in this way. For example, a single numeric value could be represented in any of the three forms.

6 Conclusion

We have described the JNET constraint reasoning system. JNET is implemented as a component of the IMAGEbot planner-based agent and it provides the planner with constraint reasoning capabilities. As a constraint reasoning system, JNET can be applied to solving constraint problems in other real-world application domains. To do so, the user needs to define variables and their domains, and specify the constraints using the predefined constraints in the constraint library. For modeling application-specific constraints that are not defined in the constraint library, JNET provides the user with two alternatives:

1. Constraints can be implemented as reusable procedural constraints by extending the constraint template provided in JNET;
2. Constraints can be implemented as a set of attached functional methods, which may be defined at runtime, without modification to, or even access to, the JNET source code.

JNET provides an easy way to integrate non-constraint-based services into a constraint-based application; any Java classes can be used as types, and any methods provided by those classes can be used to implement constraints. This capability is used in IMAGEbot to integrate planning with sensing; "sensors" that return information about a software environment, such as the locations of files, are implemented as constraint attachments; as relevant variables become constrained, different sensors (in the form of attachments) are activated, yielding additional constraints which may, in turn, activate other sensors.

Although any Java class can be used as a type, in our current implementation, not all Java *primitives* are supported. Specifically, String, boolean, long and double are supported, but other primitives, such as char and float, must be cast to long and double, respectively. Although we consider this an acceptable compromise, in future work, we will provide language-level support for primitive types, so the conversions are done automatically.

There is one domain type supported by JNET that is not currently supported by attachments: regular expressions. Infinite string domains in JNET are represented as regular languages, so attachment methods that take string arguments are limited to singletons or finite sets. Given that many commands in software environments can take regular expressions as arguments, providing the ability to specify attachments that accept regular expressions would be a natural extension, which we intend to add.

References

1. ILOG 2000. *ILOG Solver 5.0. User Manual*. ILOG, Gentilly, France, 2000.
2. N. Beldiceanu and E. Contejean. Introducing global constraints in CHIP. *Mathematical and Computer Modelling*, 20(12):97–123, 1994.
3. C. Bessiere and M. Cordier. Arc-consistency and arc-consistency again. In *Proceedings of AAAI-93*, pages 108–113, 1993.
4. A. Blum and M. Furst. Fast planning through planning graph analysis. *AIJ*, 90(1–2):281–300, 1997.
5. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman and Co., 1979.
6. K. Golden and J. Frank. Universal quantification in a constraint-based planner. In *AIPS02*, 2002.
7. Keith 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.
8. Keith Golden. Automating the processing of earth observation data. In *7th International Symposium on Artificial Intelligence, Robotics and Automation for Space*, 2003.
9. C. Green. *The application of theorem proving to question answering systems*. PhD thesis, Stanford University, 1969.
10. T. R. Jensen and B. Toft. *Graph Coloring Problems*. Wiley-Interscience, New York, 1995.
11. A. Jónsson. *Procedural Reasoning in Constraint Satisfaction*. PhD thesis, Stanford University, 1996.
12. A. Jónsson and J. Frank. A framework for dynamic constraint reasoning using procedural constraints. In *Euopean Conference on Artificial Intelligence*, 2000.
13. A. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.
14. R. Mohr and T. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.
15. U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Science*, 2:95–123, 1974.
16. B. A. Nadel. Consistent satisfaction algorithms. *Computational Intelligence*, 5:188–224, 1989.
17. R. Nemani, P. Votava, J. Roads, M. White, P. Thornton, and J. Coughlan. Terrestrial observation and predition system: Integration of satellite and surface weather observations with ecosystem models. In *Proceedings of the 2002 International Geoscience and Remote Sensing Symposium (IGARSS)*, 2002.
18. Gert Smolka. The oz programming model. In Jan van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of *Lecture Notes in Computer Science*, pages 324–343. Springer-Verlag, Berlin, 1995.
19. D. L. Waltz. Understanding line drawings of scenes with shadows. In P. H. Winston, editor, *The Psychology of Computer Vision*, pages 19–91. McGraw-Hill, 1975.
20. Monte Zweben and Mark S. Fox. *Intelligent Scheduling*. Morgan Kaufmann Publishers, San Francisco, California, 1994.