Using Object-Oriented Constraint Satisfaction for Automated Configuration Generation

Tim Hinrich, Nathaniel Love, Charles Petrie, Lyle Ramshaw, Akhil Sahai, Sharad Singhal
Internet Systems and Storage Laboratory
HP Laboratories Palo Alto
HPL-2004-141
August 17, 2004*

E-mail: {asahai@hpl.hp.com}

constraint satisfaction, configuration generation, utility computing, CIM

In this paper, we describe an approach for automatically generating configurations for complex applications. Automated generation of system configurations is required to allow large-scale deployment of custom applications within utility computing environments. Our approach models the configuration management problem as an Object-Oriented Constraint Satisfaction Problem (OOCSP) that can be solved efficiently using a resolution-based theorem-prover. We outline the approach and discuss both the benefits of the approach as well as its limitations, and highlight certain unresolved issues that require further work. We demonstrate the viability of this approach using an e-Commerce site as an example, and provide results on the complexity and time required to solve for the configuration of such an application.

* Internal Accession Date Only Approved for External Publication

Using Object-Oriented Constraint Satisfaction for Automated Configuration Generation

Tim Hinrichs, Nathaniel Love, Charles Petrie
Stanford University

Lyle Ramshaw, Akhil Sahai, Sharad Singhal
Hewlett-Packard

Abstract: In this paper, we describe an approach for automatically generating configurations for complex applications. Automated generation of system configurations is required to allow large-scale deployment of custom applications within utility computing environments. Our approach models the configuration management problem as an Object-Oriented Constraint Satisfaction Problem (OOCSP) that can be solved efficiently using a resolution-based theorem-prover. We outline the approach and discuss both the benefits of the approach as well as its limitations, and highlight certain unresolved issues that require further work. We demonstrate the viability of this approach using an e-Commerce site as an example, and provide results on the complexity and time required to solve for the configuration of such an application.

1. Introduction

Automated resource configuration has gained more importance with the advent of utility computing initiatives such as HP's Utility Data Center product [1], IBM's "on-demand" computing initiative [2], Sun's N1 vision [3], Microsoft's DSI initiative [4] and the Grid initiative [5] within the Global Grid Forum. All of these require large resource pools that are apportioned to users on demand. Currently, the resources that are available to these resource management systems are "raw" computing resources (servers, storage, or network capacity) or simple clusters of machines. The user still has to manually install and configure applications, or rely upon a managed service provider to obtain pre-configured systems from service providers.

Creating custom environments is usually not possible because every user has different requirements. Managed service providers rely on a small set of pre-built (and tested) application environments to meet user needs. However, this limits the ability of users to ask for applications and resources that have been specially configured for them. In our research, we are focusing on how complex application environments (for example, an e-Commerce site) can be automatically "built-to-order" for users. In order to create a custom solution that satisfies user requirements, many different considerations have to be taken into account. Typically, the underlying resources have technical constraints that need to be met in order for valid operations—not all operating systems will run on all processors, and not all application servers will work with all databases. In addition, system operators may impose constraints on how they desire such compositions to be created. For example, when resources are limited, only certain users may be able to request them. Finally, the users themselves have requirements on how they want the system to behave. Thus, automating the design, deployment and configuration of such complex environments is a hard problem.

2. Problem Definition

A number of languages/standards [6] [7] exist which can be used to describe resource configurations. Of these, the Common Information Model (CIM) of the Distributed Management Task Force (DMTF) [8] is widely used in the industry to represent resource configurations. CIM captures the resources as a set of classes (types) described using the Managed Object Format (MOF) representation with resource attributes described by attribute names and data types within the classes. The type model captures the resource types, and the inheritance, aggregation, and association relationships that exist between them. Instances of the classes with the attribute values filled in correspond to resource instances. Typically, the resource types deal with a large number of classes, because the models have to describe not only the "raw" resources, but also those that can be composed out of those resource types.

When resources are combined to form other higher-level resources, a variety of rules need to be followed. For example, when operating systems are loaded on a host, it is necessary to validate that the proc-

essor architecture assumed by the operating system is indeed the architecture of the host. Similarly, when an application tier is composed from a group of servers, it may be necessary to ensure that all network interfaces are configured to be on the same subnet or that the same version of the application is loaded on all machines in the tier. To ensure correct behavior of a reasonably complex application, several hundred such rules may be necessary. This is further complicated by the fact that a large fraction of these rules are not inherent to the resources, but depend on preferences (policies) provided by the system operator or indeed, by the customer as part of the request itself.

The current CIM meta model does not provide the capability to capture such rules. To accommodate these rules, we have extended the CIM meta-model to associate policies with the resource types. These policies capture the technical constraints and choices made by the operators or administrators that need to be obeyed by every instance of the associated class. An example of a resource type declaration in MOF is shown below, as well as an example of policies that could be associated with the type. The example shows how a database server might be represented using MOF. By capturing the constraints on what is possible (or permitted) for the values of the model attributes within an instance of policy that is attached to the resource type (as opposed to within the model itself), it becomes possible to customize the configurations that are valid without constantly extending the models.

```
[Version (“1.0.0”), Description (“A database server.”)]
class DatabaseServer : Resource
{
  [Description (“Type of DB server.”) ]
  String type;
};

[Association, Composition, Description (“Server type in database
server”)]
class DbServerType
{
  [Description (“Type of machine used for database server”)]
  Server REF server;
  [Description (“Database containing this machine type”]
  DatabaseServer REF database;
}

[Association, Composition, Description (“Software image in database
server”)]
class SwImageInDbServer
{
  [Description (“Software image used for Database Server”)]
  InstalledSoftware REF swImage;
  [Description (“Database containing this software image”)]
  [DatabaseServer REF database;
}

instance of ClassScopedPolicy
{
    Id = “Policy01”;
    AssociatedClasses = {“DatabaseServer“};
    Assertions = {
         “swImage.name == \”Database\””,
         “(type == \”Oracle\”) ∨ (type == \”mySQL\”)”,
         “(type == \”Oracle\”) ⇒ (swImage.version == 9)”,
         “(type == \”mySQL\”) ⇒ (server.osImage.name == \”Linux\”)”
    };
};
```

The users can request customization of particular resources from the available resource types by specifying additional constraints[1] on their attribute values and on their arrangement in the system. These requests could be for instances of “raw” resources or for composite resources. Our goal is to automatically

[1] The terms policy, constraint, and rule are frequently used interchangeably. From this point forward we will use only the term constraint.

generate a system configuration by selecting the appropriate resource classes and assigning values to their attributes so that all constraints specified in the underlying resource models are satisfied.

3. A Running Example

We will start by describing a particular utility computing problem that will be used for illustration throughout the paper. We will be using a more compact representation [9] for MOF specifications and their associated constraints. In all that follows we represent a constraint on a particular MOF specification by surrounding it with the keyword *satisfy* and including it within the specification itself.

The example in question models a collection of hardware and software components that can be assembled to build an e-Commerce site. The objects themselves can be defined hierarchically with e-Commerce at the top. An e-Commerce site includes three tiers of servers, including web, database, and applications servers; additional resources include a variety of operating systems, software applications, computers, and networking components. The class definitions in this environment contain the expected compositional constraints, like restricting mySQL to Linux servers. The example also contains mathematical constraints—resources have cost attributes with values constrained to be the sum of the costs of the objects contained within the resource. One portion of a class definition from this example—the DatabaseServer class—appears below. It is the compressed version of the example in Section 2.

```
class DatabaseServer
{
  type: String;
  server: Server;
  swImage: InstalledSoftware;
  satisfy (swImage.name == “Database”);
  satisfy ((type == “Oracle”) ∨ (type == “mySQL”));
  satisfy ((type == “Oracle”) ⇒ (swImage.version == 9));
  satisfy ((type == “mySQL”) ⇒ (server.osImage.name == “Linux”));
}
```

Here `type` is a String variable; likewise `server` and `swImage` are variables of type Server and InstalledSoftware, respectively. Note that unlike MOF, server and swImage are fields that refer to classes, while type is defined as a primitive type. Four constraints must be satisfied for all objects of type DatabaseServer. The first dictates the value of the `name` field of the `swImage` object to be “Database”. The next constraint defines a domain for the variable `type` via disjunction; it can either take on the value “Oracle” or “mySQL”. The last two constraints explain the repercussions of assigning `type` to each of the values in its domain: assigning “Oracle” to `type` means `swImage.version` must be set to 9. Assigning “mySQL” to `type` means `server.osImage.name` must be set to “Linux”. Notice how dot-notation can be used to access sub-components of non-primitive types. From the statement

```
swImage.version == 9
```

we deduce that the InstalledSoftware type must include a declaration for the variable named `version`. Likewise, Server must include a variable `osImage`, which in turn must include a variable `name`.

An instance of type DatabaseServer includes an instance of a Server and an instance of InstalledSoftware, which can be represented as shown here.

```
DatabaseServer(“mySQL”, Server(...), InstalledSoftware(...))
```

User requests in our example consist of a distinguished target class usually called main, which contains a variable of type eCommercesite. Any user requirements appear as constraints on that variable. For example, the request

```
main {
  ecomm: eCommercesite;
  satisfy (ecomm.tier1.numservers >= 10);
  satisfy (ecomm.tps == 5000);
```

```
}
```

asks for an instance of an e-Commerce site with at least ten servers in tier1, supporting 5,000 transactions per second. A solution is simply an instance of an eCommercesite object, represented just as DatabaseServer is represented above. Thus generating an e-Commerce configuration amounts to building an instance of the eCommercesite class.

The full example includes around twenty of these class definitions, ranging in complexity from an e-Commerce site down to specifications for a particular type of computer. Snippets from this problem will show up repeatedly in what follows as illustration, but the principles illustrated will be applicable to a broad range of configuration management problems.

4. Configuration Management as an OOCSP

As shown above, configuration management problems such as utility computing can often be modeled as a hierarchy of class definitions with embedded constraints. Abstracting away from the details of any particular problem can allow a more comprehensive understanding of not only the problem but also the possible routes for solution. Paltrinieri [10] outlines the notion of an Object-Oriented Constraint Satisfaction Problem (OOCSP), which turns out to be a natural abstraction for a broad class of configuration management problems. Similarly, Alloy [11] uses an object oriented specification for describing and analyzing software models.

An OOCSP is defined by a set of class definitions, a set of enumerations, and a distinguished target class, much like main in a JAVA program. Each class definition includes an ordered set of variables, each with a declared type, and a set of constraints on those variables; each class also has a name, a set of super classes, and a function Dot (.) that gives access to its variables. An enumeration is simply a set of values; declaring a variable as an enumeration forces the variable to be assigned to one of the elements in that set. A solution to an OOCSP is an instance of the target class. In an OOCSP, the constraints are embedded hierarchically so that if an object is an instance of the target class (i.e. it satisfies all the constraints within the class) it includes instances of all the target's subclasses, which also satisfy all constraints within those classes.

The OOCSP for the e-Commerce example includes class definitions for eCommercesite, DatabaseServer, Server, and InstalledSoftware among others. The class definitions contain a set of variables, each with a declared type. DatabaseServer includes (in order) a String variable `type`, a variable `server` of type Server, and a variable `swImage` of type InstalledSoftware. One of the constraints requires the name component of `swImage` to be "Database". It has no superclasses, and the function Dot is defined implicitly.

While it is clear how to declare variables within a class, many options exist for how to the express constraints on those variables. In our examples we use standard logical connectives, like $\vee$ and $\Rightarrow$, to mean exactly the same thing they do in propositional and first-order logic. We have formally defined the language chosen for representing constraints both by giving a logician a particular vocabulary and by giving a grammar; these definitions are virtually identical. The vocabulary follows, and the grammar can be found in Appendix A.

The constraint language includes all quantifier-free first-order formulas over the following vocabulary.
1. r is a relation constant iff r is the name of a class, equality or an inequality symbol
2. f is a function constant iff f is the binary Dot or a mathematical function
3. v is a variable iff v is declared as a variable or starts with a letter from the end of the alphabet, e.g. x, y, z
4. c is an object constant iff c is an atomic symbol and not one of the above

The constraints seen in the DatabaseServer example are typical and have been explained elsewhere.

Two types of constraints that do not appear in our example deserve special mention. Consider the following snippet of a class definition.

```
x: DatabaseServer;
y: DatabaseServer;
x == y;
```

We define equality to be syntactic; two objects are equal exactly when all their properties are equal. That means that two objects that happen to have all the same properties are treated as essentially the same object. The exception to this interpretation of equality is arithmetic. Not only is `7==7` satisfied, but so is `2*2==4`, as one would hope, even though syntactically 4 is different than 2*2.

The other type of notable constraint is more esoteric; consider the following.

```
x: Any;
y: Any;
satisfy (DatabaseServer("Oracle", x, y));
```

This constraint requires x and y to have values so that `DatabaseServer("Oracle", x, y)` is a valid instance of DatabaseServer. These constraints become valuable when one wants to define an object of arbitrary size, like a linked list:

```
class List {
      data: Any;
      tail: Any;
      satisfy ((tail == nil) ∨ List(tail.data, tail.tail));
}
```

This List class is recursively defined, with a base case given by the disjunct `tail == nil`; the recursive case is the second disjunct, which requires `tail` itself to be a List object. Our constraint language allows us to define these complex objects and also write constraints on those objects.

Given what it means to satisfy a constraint we can precisely describe what it means for an object to be an instance of a particular class. An instance of a class T is an ordered set of objects, one for each variable, such that (1) the object assigned to a variable of type R is an instance of R and (2) the constraints of T are satisfied. The base case for this recursive definition is the enumerations, which are effectively objects without subcomponents. Objects are instances of an enumeration if they are one of the values listed in that enumeration.

To illustrate, an instance of a DatabaseServer is an object with three components: an instance of String, an instance of Server, and an instance of InstalledSoftware. Those components must satisfy all the constraints in the DatabaseServer class. The instance of Server must likewise include some number of components that together satisfy all the constraints within Server. The same applies to InstalledSoftware.

This section has detailed how one can formulate configuration management problems as OOCSPs[2]. The next section confronts building a system to solve these configuration management problems.

5. Solving Configuration Management Problems by Solving OOCSPs

Our approach to solving configuration management problems is based on an OOCSP solver. The two main components of the system communicate through the OOCSP formalism. The first component includes a model of the utility computing environment at hand. It allows administrators to change and expand that model, and it allows users to make requests for specific types of systems without worrying too much about that model. The second component is an OOCSP solver based on a first-order resolution-style [12] theorem prover Epilog, provided by the Stanford Logic Group. It is treated as a black box that takes an OOCSP as input and returns a solution if one exists. The rest of this paper focuses on the design and implementation of the OOCSP solver and discusses the benefits and drawbacks in the context of configuration management.

[2] We believe the notion of an OOCSP is equivalent to a Context Free Grammar in which each production rule includes constraints that restrict when it can be applied.

The architecture of the OOCSP solver can be broken down into four parts. Given a set of class definitions, a set of enumerations, and a target class, a set of first-order logical sentences is generated. Next, those logical sentences are converted to what is known as clausal form, a requirement for all resolution-style theorem provers. Third, a host of optimizations are run on the resulting clauses so that Epilog can more easily find a solution. Lastly, Epilog is given the result of the third step and asked to find an instantiation of the analog of the target class. If such a solution exists, Epilog returns an object that represents that instantiation, which by necessity includes instantiations of all subcomponents of the target class, instantiations of all the subcomponents' subcomponents, and so on. Epilog also has the ability to return an arbitrary number of solutions or even all solutions. Because the conversion to clausal form is mechanical and the optimizations are Epilog-specific, we will discuss in detail only the translation of an OOCSP to first-order logic, the results of which can be used by any first-order theorem prover.

Consider the class definition for DatabaseServer. Recall we can represent an instance of a class with a term, e.g.

```
DatabaseServer("Oracle", Server(...), InstalledSoftware(...))
```

Notice this is intended to be an actual instance of a DatabaseServer object. It includes a type, Oracle, and instances of the Server class and the InstalledSoftware class. To define which objects are instances of DatabaseServer given our representation for such instances we begin by requiring the arguments to the DatabaseServer term be of the correct type.

```
DatabaseServer.instance( DatabaseServer(x, y, z) ) ⇐
      String.instance(x) ∧
      Server.instance(y) ∧
      InstalledSoftware.instance(z) ∧ ...
```

But because a DatabaseServer cannot be composed of any String, any Server instance, and any InstalledSoftware instance this sentence is incomplete. The missing portion of the rule represents the constraints that appear within the DatabaseServer class definition. These constraints can almost be copied directly from the original class definition giving the sentence shown below.

```
DatabaseServer.instance( DatabaseServer(x, y, z) ) ⇐
      (String.instance(x) ∧
       Server.instance(y) ∧
       InstalledSoftware.instance(z) ∧
       z.name == "Database" ∧
       ((x == "Oracle") ∨ (x == "mySQL")) ∧
       ((x == "Oracle") ⇒ (z.version == 9)) ∧
       ((x == "mySQL") ⇒ (y.osImage.name == "Linux")) )
```

Similar translations are done for all class definitions in the OOCSP.

Once these translations have been made for all classes and enumerations in the OOCSP to first-order logic, the conversion to clausal form is entirely mechanical and a standard step in theorem-proving. For any particular class definition these first two steps operate independently of all the other class definitions; consequently, if an OOCSP has been translated once to clausal form and changes are made to a few classes, only those altered classes must undergo this transformation again.

Once the OOCSP has been converted into clausal form the result is a set of rules that look very similar to the sentence defining DatabaseServer above. Several algorithms are run on these rules as optimizations. These algorithms prune unnecessary conjuncts, discard unusable rules, and manipulate rule bodies and heads to improve efficiency in the final step. Doing all this involves reasoning about both syntactic equality and the semantics of the object-oriented Dot function. These algorithms greatly reduce the number and lengths of the rules, consequently reducing the search space without eliminating any possible solutions. Some of these optimizations are global, which means that if any changes are made to the OOCSP those algorithms must be run again. Because one of the optimizations pushes certain types of constraints down into the hierarchy, it is especially important to apply it once a new query arrives.

The final step invokes Epilog by asking for an instantiation of the (translated) target class. If the target class were DatabaseServer, the query would ask for an instance x such that DatabaseServer.instance(x) is entailed by the rules left after optimization, i.e. x must be an instance of DatabaseServer. Moreover one can ask for an arbitrary number of these instances or even all the instances.

6. Consequences of Our Approach

We have made many choices in modeling and solving problems in the configuration management domain, both in how we represent a configuration management problem as an OOCSP and in how we solve the resulting OOCSP. This section explores those choices and their consequences.

6.1 Modeling Configuration Management Problems

The choice of the object-oriented paradigm is natural for configuration management--coupling this idea with constraint satisfaction leads to easier maintenance and adaptation of the problem so modeled. Our particular choice of language for expressing these constraints has both benefits and drawbacks and our decision to define equality syntactically may raise further questions.

Benefits

Modeling a configuration management problem as an OOCSP gives benefits similar to those gained by writing software in an object-oriented language. Class definitions encapsulate the data and the constraints on that data that must hold for an object to be an instance of the class. One class can inherit the data and constraints of another, allowing specializations of a more general class to be done efficiently. Configuration management naturally involves reasoning about these hierarchically designed objects; thus it is a natural fit with the object-oriented paradigm.

Modeling configuration management as a constraint satisfaction problem also has merits, mostly because stating a CSP is done declaratively instead of imperatively. Imperative programming requires explaining how a change in one of an object's fields must change the data in its other fields to ensure the object is still a valid instance. Doing this declaratively requires only explaining what the relationship between the fields must be for an object to be a valid instance. How those relationships are maintained is left unspecified. An imperative program describes a computational process, while the declarative version describes the results of that computational process. The extra detail an imperative object provides makes it more efficient to manipulate than its declarative equivalent, but makes maintenance and alteration more difficult.

Take a class with four fields, a, b, c, d, all of which are boolean. Say the relationship we want to express is that if a is set to true then either b or c must be set to true. If one were to write this imperatively, when a is set to true an arbitrary choice might be made between setting b or c to true. Now consider what happens when someone adds a new relationship to the fields: b is true exactly in the same cases as when d is true. If a is set to true and then b is set to true, d must also be set to true. But if d is subsequently set to false b must also be set to false, which means c must be set to true. Certainly all that could be written imperatively, but one could argue the below is easier to understand and modify:

$$a \Rightarrow b \vee c$$
$$b \Leftrightarrow d$$

Regardless whether one models objects imperatively or declaratively, both approaches require a search for a configuration that meets all such constraints. While it is unclear how one would search through a set of imperative objects to find such a configuration, solving constraint satisfaction problems has been well studied, although not all of the techniques for solving CSPs may be applicable to solving OOCSPs. The declarative language we have chosen for writing constraints, first-order logic, has the advantage that it comes equipped with powerful techniques for answering queries about sets of constraints. In the above example, a query might ask if the given sentences jointly are contradictory—do they at the same time

require `d` to be set to true and `d` to be set to false. Another query might ask for assignments of the variables that satisfy the sentences—in the above example we may set `a`, `b`, and `d` all to true. It is also noteworthy that everything that can be said in an imperative programming language can be said in first order logic, which as shown in Section 4 allows the definition of recursive classes such as linked lists. This enables the description of OOCSPs where the number of instances of a particular class is constrained by the values of other variables.

Design configuration problems have previously been addressed in three primary ways. The first is as a standard CSP problem. The OOCSP has the obvious advantage that configuration problems are easier to formulate as a set of component classes and constraints among them. In particular, a CSP requires the explicit enumeration of every possible variable that could be assigned and the OOCSP does not. For instance, suppose in the DatabaseServer class the field `swImage` can either be assigned an instance of InstalledSoftware or an instance of InstalledServerSoftware because of inheritance. Suppose InstalledSoftware has a single field `package` and that InstalledServerSoftware has both a `package` field and a `license` field. A CSP formulation must explicate a variable corresponding to the possibility of `swImage.license` with a domain that includes the legitimate enumerations of `license` field as well as a special null value. Otherwise the CSP cannot account for the possibility of a solution with a choice of class InstalledServerSoftware.

Design configuration has also been attempted with expert systems [13] but domain knowledge rules are too difficult to manage because of implicit control dependencies, so the approach does not scale. The OOCSP has the advantage that the formalism is clear and the ordering of the domain knowledge has no impact on the set of possible solutions. A third approach has been to add search control as heuristics to a structure of goals and constraints [14] [15], but this approach is more complex and slower than the OOCSP approach.

Limitations

The choices outlined above do have drawbacks. In particular first-order logic is very expressive, so using it as our constraint language comes at a cost: first-order logic is fundamentally undecidable—there is no algorithm that can ensure it will always give the correct answer and at the same time halt on all inputs. If there is a solution it will be found in a finite amount of time; otherwise the algorithm may run forever. We have not yet determined the decidability and complexity of the subset of first-order logic we are using in our research. Simpler languages might lead immediately to certain complexity bounds, but as mentioned above we are interested in solving problems where we are selecting both the classes that need to be instantiated, as well as the number of instances of those classes based on arbitrary constraints. We have chosen to start with a language expressive enough to write such constraints, and we can restrict if further if decidability or complexity become practical issues for particular applications.

The OOCSP described in this paper makes a commitment to a particular notion of equality–two objects are equal whenever they are syntactically equal. The constraint

```
x.computer == y.computer
```

ensures the variables x and y are assigned instances such that their computers have all the same properties. Thus two objects are equal exactly when all their fields are equal. More importantly two objects that happen to have the same properties are treated as the same object; thus, there is no way to write the constraint that two objects have the same properties but are different objects. This becomes troublesome when one introduces resource pools and needs to know the number of servers that must be allocated for a particular configuration. Under some mild assumptions, in our implementation instances of objects always begin with unique attribute values; they are set to something different only if constraints require it. Giving each object an extra field of type Any (perhaps called ID) thus resolves this ambiguity.

These design decisions could have been made differently. The object-oriented approach seems a good fit, and using a declarative language is not a surprising choice. The use of first-order logic could be criticized as overkill, though its expressiveness does allow one to create more complex types of objects, like lists and trees. The commitment to syntactic equality is probably the most questionable but may be sufficient for the purposes of solving a broad category of configuration management problems.

6.2 Solving OOCSPs by Translation to First-Order Logic

Once a configuration problem has been modeled as an OOCSP, several options are available for building a configuration that meets the requirements embedded in that OOCSP. We have chosen to find such configurations by first translating the OOCSP into first-order logic sentences and then invoking a resolution-based theorem prover. To rehash the system's architecture, the input to the system is an OOCSP. That input is first translated into first-order logic, which is in turn translated to a form suitable for resolution-style theorem provers; this form is then optimized for execution in Epilog.

Benefits

Translating an OOCSP into first order logic can be done very quickly, in time linearly proportional to the number of class definitions. Both this translation and the one from first-order logic to clausal form can be performed incrementally; each class definition is translated independently of the others. The bulk of the optimization step can also be run as each class is converted, but the global optimizations can be run only once the user gives the system a particular query. These optimizations aggressively manipulate the set of constraints so it is tailored for the query at hand.

Using Epilog as the reasoning engine provides capabilities common to first-order theorem provers. Epilog can both produce one answer and all answers. More interestingly it can produce a function that with each successive call returns a new solution, giving us the ability to walk through as much or as little of the search space as needed to find the solution we desire. As we will discuss in Section 7, Epilog can at times find solutions very rapidly.

Limitations

While the translation from an OOCSP into first-order logic requires time linearly proportional to the size of the OOCSP, our use of a resolution-based theorem prover requires those first-order sentences be converted into clausal form. There may be an exponential increase in the number of sentences when doing this conversion; thus not only the time but also the space requirements can become problematic.

Another source of discontent is the number of solutions found by Epilog. Many theorem provers treat basic mathematics, addition, multiplication, inequality, etc., with procedural attachments. This means that if one of the constraints requires `x < 5`, the theorem prover will find solutions only in those branches of the search space where x is bound to a number that happens to be less than five. If x is not assigned a value the theorem prover will not arbitrarily choose one for it. Our theorem prover, Epilog, has these same limitations. Thus the version of Epilog we have been discussing will find no solutions to the following OOCSP, where the target class is main.

```
class main {
    x: int;
    satisfy (x < 5);
}
```

While clearly we should get some solution for this example, we do not want infinitely many solutions. A more computationally expensive version of Epilog has the ability to return an instance of the target class and conditions under which that instance is valid. In this case Epilog would return

```
main(y), y < 5
```

We have not yet experimented with this capability, but it is the subject of future work.

Yet another problem with using first-order logic is derived from one of the benefits mentioned in Section 6.1. It is as expressive as any programming language, i.e. first-order logic is Turing complete. That means answering queries about a set of first-order sentences is formally undecidable; if the query can be answered positively, Epilog will halt. If the query cannot be answered positively Epilog may run forever.

This problem is common to all algorithms and systems that soundly and completely answer queries about first-order sentences. But it seems undecidability may also be a property of OOCSPs; our conversion to first order logic may not be overcomplicating the problem of finding a solution at all. Theoretically our approach to solving OOCSPs may turn out to be the right one; however, from a pragmatic standpoint many OOCSPs will simply be hierarchical representations of CSPs, which means such OOCSPs are decidable.

7. Experimental Results and Future Work

The OOCSP solver architecture is a fairly simple one, and for our running example results are promising, even at this early stage. Translating the OOCSP with eighteen classes into clausal form requires four to five minutes and results in about 1150 rules. The optimization process finishes in five seconds and reduces the rule count to around 620. Those eighteen class definitions and the user request allow for roughly 150 billion solutions; in other words, our example is underconstrained. That said, Epilog finds the first solution in 0.064 seconds; it can find 39000 solutions in 147 seconds before filling 100 MB of memory, which is a
rate of 1000 solutions every 3-4 seconds. If we avoid the memory problem by not storing any solutions but only walking over them, it takes 114 seconds to find those same 39000 answers--the number of answers returned by Epilog is entirely up to the user. These are results for a single example. More complicated examples are the subject of future work[3].

The limitations discussed in Section 6 present a host of problems: possible undecidability, exponential blowup when converting to clausal form, inexpressiveness of syntactic equality, incompleteness of mathematical operators. Undecidability might be dealt with by restricting the constraint language significantly. Clausal form is fundamental to using a resolution-based theorem prover; changing it to eliminate the accompanying conversion cost would require building an entirely new system. Syntactic equality, while less expressive than we might like, may be sufficient for solving the class of problems we want to solve. As mentioned in Section 6.2, entirely fixing the incompleteness of mathematical operators may do more harm than good, and finding a balance between too many solutions and too few will be key to success when iterating over solutions.

The system configuration problem, however, is not the only problem to be solved when building an automatic configuration management service. In order to use one of the configurations the system has produced, that configuration must be coupled with a workflow—a structured set of activities—that will bring the configuration on line [16].

Deductive plan synthesis deals with constructing workflows (a.k.a. plans) by carefully writing logical constraints. One must describe a set of properties that describe the world, a set of actions that change those properties, a description of the current state of the world, and a description of the desired state of the world. Once these have been written as logical sentences, one need only ask the proper query to produce a workflow that achieves the desired state of the world. We have described here John McCarthy's situation calculus [17], which has been explored and expanded for 35 years. The convenient part is that an OOCSP is expressive enough to embed these carefully crafted sentences. Thus one need only write the correct OOCSP to produce both a configuration and a workflow. We are currently investigating this idea.

Once we have a system that automatically produces a configuration and a workflow, we plan to extend the system to adjust incrementally to shifting resource demands or possible component failures. In the utility computing domain, an e-Commerce site might be receiving more traffic than was initially expected; the configuration must be changed to accommodate more transactions per second. This requires not only a new configuration to be built (ideally with as few changes as possible), but also a workflow that alters the current configuration to the new one. And just as in the configuration problem, a customer or system administrator might want to choose from among several options.

[3] These statistics are for a 500 MHz PowerPC G4 processor with 1 GB of RAM and Epilog running on MCL 5.0.

8. Conclusion

In this paper, we have described an approach to automated configuration management that relies on an Object-Oriented Constraint Satisfaction Problem (OOCSP) formulation. By posing the problem as an OOCSP, we can specify system configuration in a declarative form and apply well-understood techniques to rapidly search for a configuration that meets all specified constraints. We discussed both the benefits and limitations of this approach.

We are currently evaluating other examples to understand how this approach scales as both the problem size and the number of constraints on the problem increases but in our initial experiments, we have found that it is feasible for small to medium sized problems. We are also exploring how our approach can be used to automatically generate both a configuration that satisfies all given constraints, as well as a workflow that can be used to instantiate that configuration.

References

1. HP Utility Data Center (UDC) http://www.hp.com/solutions1/infrastructure/solutions/utilitydata
2. IBM Autonomic Computing http://www.ibm.com/autonomic
3. SUN N1 http://wwws.sun.com/software/solutions/n1/
4. Microsoft DSI http://www.microsoft.com/management/
5. Global Grid Forum http://www.ggf.org
6. Unified Modeling Language (UML) http://www.uml.org/
7. SmartFrog http://www.smartfrog.org/
8. CIM http://www.dmtf.org/standards/cim/
9. A. Sahai, S. Singhal, R. Joshi, V. Machiraju, "Automated Policy-Based Resource Construction in Utility Environments" *Proceedings of the IEEE/IFIP Network Operations and Management Symposium*, Seoul, Korea, Apr. 19-23, 2004
10. M. Paltrinieri, "Some Remarks on the Design of Constraint Satisfaction Problems," *Second International Workshop on the Principles and Practice of Constraint Programming*, pp. 299-311, 1994.
11. Alloy http://sdg.lcs.mit.edu/alloy/
12. J. A. Robinson, "A machine-oriented logic based on the resolution principle," *Journal of the Association for Computing Machinery*, 12:23-41, 1965.
13. M. R. Hall, K. Kumaran, M. Peak, and J. S. Kaminski, "DESIGN: A Generic Configuration Shell," *Proceedings 3rd International Conference on Industrial & Engineering Applications of AI and Expert Systems*, 1990.
14. S. Mittal and A. Araya. "A Knowledge-Based Framework for Design," *Proceedings of the 5th AAAI*, 1986.
15. C. Petrie, "Context Maintenance," *Proceedings AAAI-91*, pp. 288-295, 1991.
16. A. Sahai, S. Singhal, R. Joshi, V. Machiraju, "Automated Generation of Resource Configurations through Policy," to appear in *Proceedings of the IEEE 5th International Workshop on Policies for Distributed Systems and Networks*, YorkTown Heights, NY, June 7-9, 2004
17. J. McCarthy and P. J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence 4*, pp. 463-502, 1969.

Appendix A: Constraint Language Grammar

```
<implication> ::=
          <or> ⇔ <implication> |
          <or> ⇐ <implication> |
          <or> ⇒ <implication> |
          <or>

<or> :=
     <and> ∨ <or> |
```

```
    <and>

<and> :=
    <neg> ∧ <and> |
    <neg>

<neg> := ¬ <atom> | <atom>
<atom> := <simple-atom> | <class-atom>
<simple-atom> :=
           <term> == <term> |
           <term> < <term> |
           <term> > <term>

<class-atom> := <class-name>(<term>, . . . ,<term>)

<term> := <dot-term> | <simple-term>
<dot-term> := <dot-term>.<field-name> | <variable>
<simple-term> :=
           <simple-term><mathematical-operator><simple-term> |
           <variable> |
           <atomic-symbol>

<variable> := <lower case symbol from end of alphabet or declared variable>
<field-name> := <atomic-symbol>
<mathematical-operator> := +| - | * | /
```