Artificial Intelligence for Engineering Design, Analysis and Manufacturing (2011), *25*, 163–174.
© Cambridge University Press 2011 0890-0604/11 $25.00
doi:10.1017/S0890060410000600

Reasoning about conditional constraint specification problems and feature models

RAPHAEL FINKEL[1] AND BARRY O'SULLIVAN[2]
[1]Department of Computer Science, University of Kentucky, Lexington, Kentucky, USA
[2]Cork Constraint Computation Centre, University College Cork, Cork, Ireland

(RECEIVED March 19, 2010; ACCEPTED October 29, 2010)

Abstract

Product configuration is a major industrial application domain for constraint satisfaction techniques. Conditional constraint satisfaction problems (CCSPs) and feature models (FMs) have been developed to represent configuration problems in a natural way. CCSPs are like constraint satisfaction problems (CSPs), but they also include potential variables, which might or might not exist in any given solution, as well as classical variables, which are required to take a value in every solution. CCSPs model, for example, options on a car, for which the style of sunroof (a variable) only makes sense if the car has a sunroof at all. FMs are directed acyclic graphs of features with constraints on edges. FMs model, for example, cell phone features, where utility functions are required, but the particular utility function "games" is optional, but requires Java support. We show that existing techniques from formal methods and answer set programming can be used to naturally model CCSPs and FMs. We demonstrate configurators in both approaches. An advantage of these approaches is that the model builder does not have to reformulate the CCSP or FM into a classic CSP, converting potential variables into classical variables by adding a "does not exist" value and modifying the problem constraints. Our configurators automatically reason about the model itself, enumerating all solutions and discovering several kinds of model flaws.

Keywords: Alloy; Answer-Set Programming; Configuration; Constraint Satisfaction; Flaw Detection

1. INTRODUCTION

Product configuration has provided constraint programming with one of its most successful application domains (Sabin & Weigel, 1998; Junker, 2006). Model-based, particularly constraint-based, approaches to configuration are the most successful in practice (http://www.gartner.com), because constraint-based product configurators are specified in a highly declarative formalism.

Configuration presents several modeling and reasoning challenges. First, it is challenging to maintain consistent integration between product catalogs and constraint-based configurator models. Second, constraint-based approaches need to be able to handle taxonomic inheritance among components and subsystems. Third, the space of possible configurable products is often unbounded but might be subject to resource restrictions. Fourth, users have preferences, and full customization must be possible.

Reprint requests to: Raphael Finkel, Department of Computer Science, University of Kentucky, Lexington, KY 40506, USA. E-mail: raphael@cs.uky.edu

Most configurator engines restrict the configuration process to some degree. In particular, a configurator will typically configure systems before subsystems. Also, isomorphic configurations provide challenges for the configurator; isomorphic configurations can be regarded as being structurally symmetric. For this reason, many configurators represent the product being configured as a set of systems rather than associating each system with a variable, which can introduce unnecessary symmetries into the configuration space.

Although constraint satisfaction techniques have supported configuration for many years, they have required extensions to the basic constraint satisfaction problem. For example, *composite* constraint satisfaction (Sabin & Freuder, 1996) has been introduced to handle hierarchical system configuration. Mittal and Falkenhainer (1990) introduced *dynamic* constraint satisfaction to cover problems in which the existence of features depended on the existence or values of other features; this scheme was subsequently given a formal logical semantics (Bowen & Bahler, 1991). This work was subsequently extended by other researchers (Mailharro, 1998; Stumptner et al., 1998). Dynamic constraint satisfaction has more recently been referred to as *conditional* constraint satisfaction (Gelle & Faltings, 2003) to distinguish dynamism due to conditional relevance of some variables and constraints from dynamism due to uncertainty and environmental change.

Many authors (including Mittal and Falkenhainer) reformulate conditional constraint satisfaction problems (CCSPs) into classic CSPs by introducing redundant domain values and augmenting the problem constraints so that some problem variables take a "not defined" value (Sabin & Gelle, 2006). Although feature models (FMs) appear quite different from CCSPs, they can also be mapped to CSPs (Benavides et al., 2005) and other data structures. These reformulations are problematic. First, they seem unnatural as a modeling technique, especially for large real-world configuration problems. Second, they become impractical and difficult to maintain, especially when the configuration space is extremely large or unbounded.

Our motivation arises from sophisticated tools that the formal methods community has developed for modelling and reasoning about complex engineered artifacts that can be regarded as configuration problems (Hinchey et al., 2008). Our objective is to study the utility of formal methods for modeling and reasoning about configuration models. The two main contributions of this paper are the following:

1. Using well-known examples, we show how to model constraint-based configuration problems naturally and concisely in the formal methods package Alloy (Jackson, 2002), which is usually used for modeling software systems, and in the answer-set programming (ASP) language *lparse* (http://www.tcs.hut.fi/Software/smodels/)
2. In addition to providing a natural modeling paradigm, these approaches are capable of providing reasoning capabilities that are very appropriate for configuration, in particular, *verifying the specification* of the configuration problem to ensure that specific flaws are absent, a problem identified and studied in earlier work (Sabin & Freuder, 1998). We argue that formalisms such as Alloy and *lparse* provide modeling tools that can be easily used by nonexperts to model and reason about configuration problems directly and naturally.

In Section 2 we informally present conditional constraint satisfaction, motivated by a well-known configuration problem, which we use as a running example. We also list some flaws that can occur in the specification of conditional configuration problems. We present both a formal methods approach (Section 3) and an answer set programming approach (Section 4) to reasoning about CCSPs. In Section 5 we show how we can easily identify flaws in CCSPs and demonstrate that Mittal and Falkenhainer's benchmark problem exhibits such flaws. We briefly show how ASP can easily find solutions involving the minimum or maximum number of options in Section 6. We turn our attention to FMs in Section 7, showing how an ASP approach can reason about these configuration models. In Section 8 we present XML representations for both CCSPs and FMs and report on our software to apply ASP methods to the data stored in such XML representations. Finally, in Section 9 we draw several conclusions and summarize our plans for future study.

2. CONDITIONAL CONSTRAINT SATISFACTION

Mittal and Falkenhainer (1990) introduced CCSPs. A CCSP differs from a classical CSP in that some variables are marked as **potential**, which means that they need not take a value in all solutions. CCSPs allow activity constraints that deal with the existence of **potential variables**, including the following:

- **require variable (RV)**, which stipulates that under certain value assignments to other variables, a potential variable must exist;
- **require not variable (RN)**, which stipulates that under certain value assignments to other variables, a potential variable must not exist;
- **always require variable (ARV)**, which stipulates that the existence of some other variable implies the existence of a potential variable; and
- **always require not variable (ARN)**, which stipulates that the existence of some other variable precludes the existence of a potential variable.

Mittal and Falkenhainer demonstrate these concepts by presenting two examples. In the first, the task is to generate valid configurations of options for a car. Because we plan to encode this example for our own purposes, we present it essentially as Mittal and Falkenhainer do in Figure 1. This small model captures, among other constraints, that luxury vehicles must have some sort of sunroof (constraint 1), that any sort of sunroof requires an option for glass (constraint 6), that an `sr1` sunroof has no opener (constraint 10), and that a luxury car may not have an `ac1` air conditioner (constraint 14).

Given such a CCSP, one can pose several queries:

1. Find/count/enumerate solutions to the CCSP. To **find** is to compute a single solution; to **count** is to discover the number of unique solutions, and to **enumerate** is to list all those solutions.
2. Enumerate all variable flaws in the CCSP. A **variable flaw** is a potential variable that is present in all solutions, so it is really a classical, not a potential, variable, or a potential variable that is never present in any solution.
3. Enumerate all value flaws in the CCSP. A **value flaw** is a value for a variable (actual or potential) that is never achieved by any solution.
4. Find/count/enumerate minimum/maximum solutions to the CCSP. A **minimum (maximum) solution** is one with the fewest (most) potential variables.
5. Find/count/enumerate minimal/maximal solutions to the CCSP. A **minimal (maximal) solution** is one in which removing (adding) any potential variable leads to a nonsolution.

classical variables

`Package`	`{luxury, deluxe, standard}`
`Frame`	`{convertible, sedan, hatchback}`
`Engine`	`{small, med, large}`

potential variables

`Battery`	`{small, med, large}`
`Sunroof`	`{sr1, sr2}`
`AirConditioner`	`{ac1, ac2}`
`Glass`	`{tinted, notTinted}`
`Opener`	`{auto, manual}`

activity constraints

1. $\texttt{Package} = \texttt{luxury} \overset{RV}{\Rightarrow} \texttt{Sunroof}$
2. $\texttt{Package} = \texttt{luxury} \overset{RV}{\Rightarrow} \texttt{AirConditioner}$
3. $\texttt{Package} = \texttt{deluxe} \overset{RV}{\Rightarrow} \texttt{Sunroof}$
4. $\texttt{Sunroof} = \texttt{sr2} \overset{RV}{\Rightarrow} \texttt{Opener}$
5. $\texttt{Sunroof} = \texttt{sr1} \overset{RV}{\Rightarrow} \texttt{AirConditioner}$
6. $\texttt{Sunroof} \overset{ARV}{\Rightarrow} \texttt{Glass}$
7. $\texttt{Engine} \overset{ARV}{\Rightarrow} \texttt{Battery}$
8. $\texttt{Opener} \overset{ARV}{\Rightarrow} \texttt{Sunroof}$
9. $\texttt{Glass} \overset{ARV}{\Rightarrow} \texttt{Sunroof}$
10. $\texttt{Sunroof} = \texttt{sr1} \overset{RN}{\Rightarrow} \texttt{Opener}$
11. $\texttt{Frame} = \texttt{convertible} \overset{RN}{\Rightarrow} \texttt{Sunroof}$
12. $\texttt{Battery} = \texttt{small}\ \&\ \texttt{Engine} = \texttt{small} \overset{RN}{\Rightarrow} \texttt{AirConditioner}$

classical constraints

13. $\texttt{Package} = \texttt{standard} \Rightarrow \texttt{AirConditioner} \neq \texttt{ac2}$
14. $\texttt{Package} = \texttt{luxury} \Rightarrow \texttt{AirConditioner} \neq \texttt{ac1}$
15. $\texttt{Package} = \texttt{standard} \Rightarrow \texttt{Frame} \neq \texttt{convertible}$
16. $\texttt{Opener} = \texttt{auto}\ \&\ \texttt{AirConditioner} = \texttt{ac1} \Rightarrow \texttt{Battery} = \texttt{med}$
17. $\texttt{Opener} = \texttt{auto}\ \&\ \texttt{AirConditioner} = \texttt{ac2} \Rightarrow \texttt{Battery} = \texttt{large}$
18. $\texttt{Sunroof} = \texttt{sr1}\ \&\ \texttt{AirConditioner} = \texttt{ac2} \Rightarrow \texttt{Glass} \neq \texttt{tinted}$

Fig. 1. A car-configuration problem based on Mittal and Falkenhainer (1990).

3. REASONING ABOUT CCSPs IN ALLOY

Alloy Analyzer 4.0 is a language originally intended to model design of data structures. Jackson presents the formal semantics of Alloy in a comprehensive manner (Jackson, 2002). Alloy has been widely used for modeling large complex engineering systems (http://alloy.mit.edu/community/models). It provides a way to specify types and constrain their instances. It can convert those types and constraints into SAT problems that it then solves, displaying the solutions via a graphical interface. If it fails to find a solution, the specification is most likely inconsistent, although the solver might not have searched a large enough population of instances; the specification indicates how many instances of each type to generate for testing purposes. In this sense, Alloy is not a complete solver.

If the graphical representation of the solution seems erroneous to the user, new constraints that the user adds to the specification can prevent the erroneous interpretation.

We find that Alloy is well suited to represent CCSPs. Figure 2 presents our Alloy representation of part of the car example from Figure 1. In Alloy, a **`sig`** introduces a type. These types, something like classes in object-oriented programming languages, may be defined to contain members. To model the car-configuration problem, we introduce a **`sig`** called `Car` with a member for each variable. During configuration we define instances of `Car`.

Each car has required attributes, including a package. The fact that every instance of a car comprises one of each of these attributes is specified with the keyword **`one`**. These attributes represent the *classical* CSP variables of the problem.

A car has additional optional attributes, including a battery and an air conditioner. These attributes correspond to the *potential* variables of the problem. We specify them with the keyword **`lone`** to state that each instance of a car may have *at most one* instance of each of these attributes. Alloy can now generate one instance of every classical variable and

```
sig Car {
 package:one Package,
 battery:lone Battery,
 ac:lone AirConditioner,
 glass:lone Glass,
 sunroof:lone Sunroof
}
abstract sig Package {}
lone sig Luxury, Deluxe, Standard extends Package {}
abstract sig Battery {}
lone sig BSmall, BMedium, BLarge extends Battery {}
abstract sig AirConditioner {}
lone sig AC1, AC2 extends AirConditioner {}
abstract sig Opener {}
lone sig Auto, Manual extends Opener {}
fact {
 all c:Car | c.package in Luxury => one c.ac // RV 2
 all c:Car | one c.sunroof => one c.glass // ARV 6
 all c:Car | one c.sunroof and c.sunroof in SR1 => no c.opener // RN 10
 all c:Car | c.package in Standard => no (c.ac & AC2) // Classical 13
 all c:Car | c.package in Luxury => no (c.ac & AC1) // Classical 14
}
run {} for 1
```

Fig. 2. An Alloy model encoding the car-configuration problem (excerpt).

an optional instance of every potential variable. The particular instance that Alloy generates captures the CSP idea of a variable's value.

We introduce each CCSP variable with an **abstract sig**, introducing a type (such as `Package`) that has no direct instances. Then we introduce subtypes (such as `Luxury`). These subtypes may have at most one instance each.

Constraints are represented inside a **fact**. RV and ARV constraints differ in the form of their left-hand side, referring either to values (like `Luxury` in constraint 2) or variables (like `sunroof` in constraint 6). RN and ARN constraints (like constraints 13 and 14) differ from RV and ARV constraints only in that they have **no** on the right-hand side.

This representation would lead to a fifth and sixth sort of constraint not contemplated by Mittal zFalkenhainer, in which the nonexistence of a potential variable leads to the existence or nonexistence of another potential variable. We would model such constraints in Alloy by facts with **no** on the left-hand side and either **one** or **no** on the right-hand side.

The Alloy program is executable. It generates the solution in Figure 3, among others. Unfortunately, Alloy gives us no way to directly count or enumerate the solutions, short of interacting multiple times with the Alloy Analyzer to request the next solution.

4. REASONING ABOUT CCSPs IN ASP

To represent CCSPs using ASP, we use the syntax that *lparse* recognizes and converts to a form acceptable to solvers such as *smodels* (Niemela & Simons 1997), *clasp* (Gebser et al., 2007), and *Cmodels* (which converts *lparse* into SAT and invokes a SAT solver; Giunchiglia et al., 2004). ASP programs deal with **predicates**, which are either true or false. We introduce a predicate for each value of each actual and potential variable. For instance, the predicate `package(luxury)` represents the value `luxury` for the variable `package`. In any given model, this predicate is either true or false.

Many ASP solvers allow **cardinality-constrained predicates**, in which the number of true predicates in a given list is bounded above, below, or both. We say

```
0 { battery(bsmall), battery(bmed),
    battery(blarge) } 1
```

to represent a cardinality-constrained predicate stating that at least 0 and at most 1 of the three predicates in the list is true. The car may have no battery at all, but if it has one, the battery must be one of small, medium, or large. *lparse* allows a shorthand for lists of predicates that share the same functor; we can equivalently write

```
0{ battery(bsmall; bmed; blarge) } 1
```

We use ASP **implications** to represent CCSP constraints:

```
1 { battery(bsmall; bmed; blarge) }
  :- package(luxury).
```

This implication says that if a car has the luxury package, it must have at least one of the battery sizes.

Finally, ASP programs have **failures**,[1] indicated by an empty left-hand side of an implication. The conjunction of

[1] The ASP community calls them *constraints*, but we avoid that term here because it conflicts with CSP terminology.

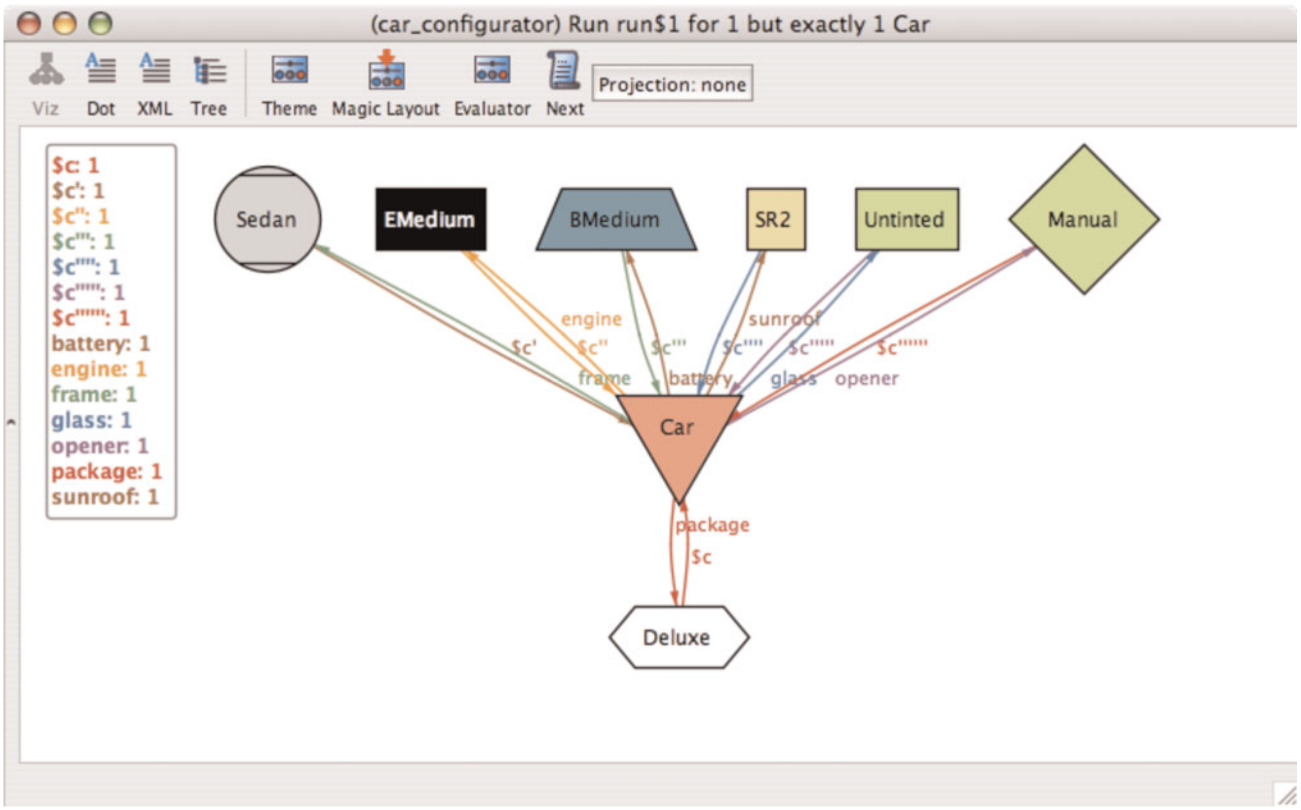

Fig. 3. A car configuration comprising a deluxe package, a sedan frame, a medium engine, a medium battery, sunroof `SR2` with untinted glass, and a manual opener. [A color version of this figure can be viewed online at journals.cambridge.org/aie]

predicates (some of which may be negated) on the right-hand side must not be true in any satisfying model. For example,

```
:- 1 { opener(auto; manual)}, sunroof(sr1).
```

specifies that no model may have an `sr1` sunroof and have either an automatic or a manual opener.

An excerpt of the cars specification in *lparse* syntax is presented in Figure 4. Although the *lparse* representation is not as elegant as the Alloy one, it is not difficult to read. Instead of **`one`**, we bound above and below by 1 (as in the rule for the classical variable `package`). Instead of **`lone`**, we bound below by 0 and above by 1 (as in the rule for the potential variable `battery`). Instead of **`no`**, we bound a failure below by 1, as in rule 10. We represent constraints that preclude particular values by failures (rule 13). We represent classical constraints that imply particular values by implications (rule 16).

ASP solvers, unlike Alloy, can easily enumerate all solutions. Each solution is an answer set, that is, a set of predicates that satisfies all the rules in the specification. A few of the 450 solutions to the car configuration specification are presented in Table 1. One can look through the list of solutions to search for variable and value flaws. However, one can also generate them automatically, as we show in the next section.

5. AUTOMATICALLY CHECKING FOR SPECIFICATION FLAWS

Specifications of CCSPs can contain a variety of flaws that can be difficult to detect manually (Sabin & Freuder, 1998).

```
1 {package(luxury; deluxe; standard)} 1. % classical variable
0 {battery(bsmall; bmed; blarge)} 1.     % potential variable
0 {airConditioner(ac1; ac2)}       1.    % potential variable
1 {airConditioner(ac1; ac2)} :- package(luxury). % RV #2
1 { glass(tinted; untinted)} :- 1 { sunroof(sr1; sr2))}. % ARV #6
:- 1 { opener(auto; manual)}, sunroof(sr1). % RN #10
:- package(standard), airConditioner(ac2). % classical #13
battery(bmedium) :- opener(auto), aircon(ac1). % classical #16
```

Fig. 4. The car-configuration problem implemented in *lparse* (excerpt).

Table 1. *A sample of solutions from the ASP model*

Pack	Frame	Engine	Battery	Sunroof	AC	Glass	Opener
`Standard`	`Sedan`	`esmall`	`blarge`	`sr2`	—	—	`Auto`
`Standard`	`Hatch`	`esmall`	`bsmall`	—	—	—	—
`Deluxe`	`Hatch`	`esmall`	`bmed`	`sr1`	`ac1`	`Tinted`	—
`Deluxe`	`Hatch`	`esmall`	`bsmall`	`sr2`	—	`Not`	`Manual`

Note: ASP, answer-set programming.

We can apply both the Alloy and ASP approaches to identify flaws in the constraint specification. In particular, we can discover that the car-configuration problem exhibits both a variable flaw and a value flaw. As we mentioned earlier, a **variable flaw** occurs when a potential variable is required in all models, that is, an option is not really optional. A **value flaw** occurs when a value cannot exist in any valid configuration, so it does not represent an option.

5.1. Checking for specification flaws in alloy

We extend the model we presented in Figure 2 to introduce an abstract type `Flaw`, with **`lone`** subtypes for each category of flaw for we would like to test. Here are some of the **`sig`** definitions for the possible flaws in our model.

```
abstract sig Flaw {}
lone sig
  noLuxury, noDeluxe, noStandard,
  noBSmall, noBMedium, noBLarge, batteryFlaw,
  noAC1, noAC2, ACFlaw
  } extends Flaw {}
```

We then introduce constraints that force the existence of flaw instances, such as one for `BatteriesFlaw`.

```
fact { one noBSmall iff no BSmall }
fact { one batteryFlaw
       iff (no c: Car| no c.battery) }
```

Given similar definitions for each flaw, we can run the Alloy Analyzer requiring no flaws for a large number of cars as follows:

```
run {} for 4 but exactly 4 Car, 0 Flaw
```

This run fails to find an instance; by experiment, we need to raise the number of flaws to 2 before the analyzer finds a solution. This solution includes an instance of `batteryFlaw` (all cars have batteries: a variable flaw) and `noConvertible` (there are no convertibles: a value flaw).

Figure 5 shows the Alloy Analyzer visualization of the flaws in Mittal and Falkenhainer's specification from Figure 1. We clearly see four instances of `Car`, with links to their associated components. However, on the far right of the figure we see an instance of `batteryFlaw` and of `noConvertible`. The fact that we see instances of these flaws demonstrates that they exist in the specification. The instance of `batteryFlaw` indicates the presence of the variable flaw highlighted above, namely, that batteries are not optional, despite the fact that the specification suggests the opposite. The presence of the value flaw that no convertible cars are possible is indicated by the instance of `noConvertible`.

We can explain these flaws, once we find them, by referring to the original specification of Figure 1. Rule 7 forces a battery in every car that has an engine, and `engine` is a classical variable. We might as well call `battery` a classical variable as well.

The other, `noConvertible`, is a value flaw: it is impossible to generate a convertible. This flaw is hidden in the implications of the activity and classical constraints. By constraint 11, convertibles do not have sunroofs. By constraints 1 and 3, cars with the luxury and deluxe packages do have sunroofs, so by elimination, convertibles must have the standard package. But by rule 15, cars with the standard package are not convertibles.

5.2. Checking for specification flaws in ASP

We expand the *lparse* representation for the car CCSP by adding a second, numeric, argument to every predicate. The new argument represents car number. For example, `package (luxury,4)` is a predicate indicating that the fourth car has a luxury package. Now rules like

```
1 {opener(auto,N), opener(manual,N)}
  :- sunroof(sr2,N) .
```

are shorthands that *lparse* expands (in a process called **grounding**) to a new rule for each valid value of `N`. We can limit any solution to four car designs.

```
number(1..4).
#domain number(N).
```

The number 4 is arbitrary; we will use it for the examples to follow. Next, we introduce new nullary predicates for each value of each variable (both classical and potential) to indicate the fact that no car at all uses a particular value, such as in this rule:

```
noLuxury :- {package(luxury,M):number(M)} 0.
```

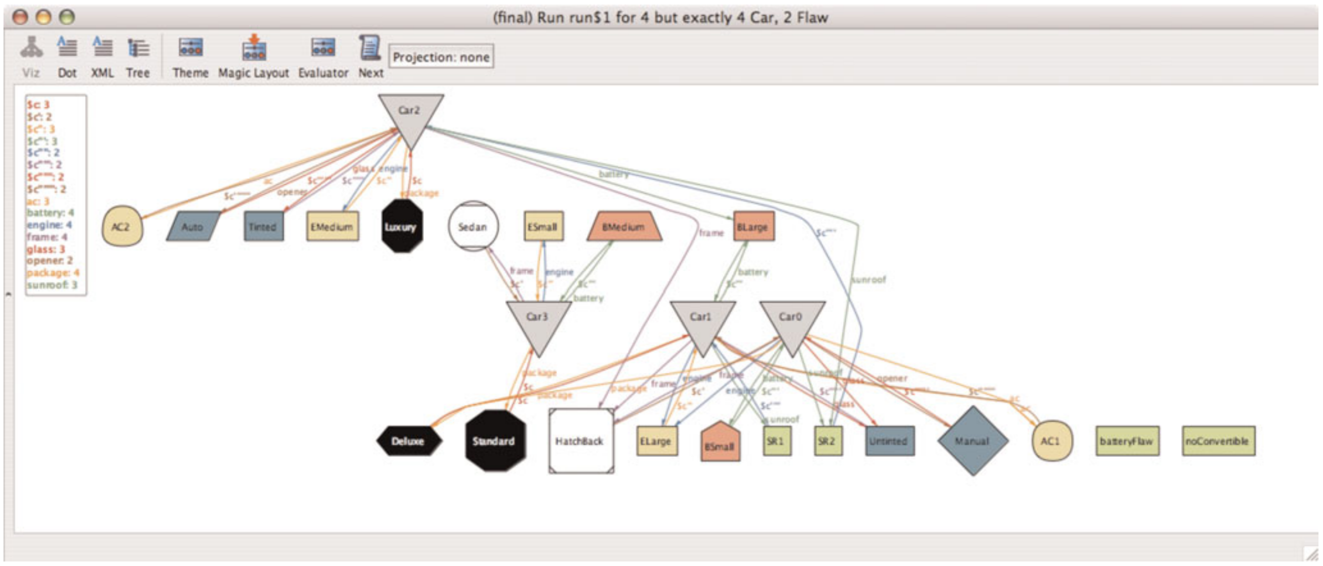

Fig. 5. The Alloy Analyzer visualization of the flaws in Mittal and Falkenhainer's (1990) specification from Figure 1. We clearly see four instances of `Car`, with links to their associated components. However, on the far right of the figure we see instances of `batteryFlaw` and `noConvertible`. That we see instances of these flaws demonstrates that they exist in the specification. [A color version of this figure can be viewed online at journals.cambridge.org/aie]

The grounder converts this shorthand into a rule containing a list of predicates `package(luxury,1)` to `package(luxury,4)`. If not a single one of these `package` predicates is true, which happens if none of the N cars has the luxury package, then `noLuxury` is true, indicating a possible value flaw.

For each potential variable, we introduce two rules, like these:

```
okSunroof :- {sunroof(sr1,N), sunroof(sr2,N)}0.
sunroofFlaw :- not okSunroof.
```

The grounder expands the first rule to four rules, one for each car. If for any car, no sunroof at all is specified, then `okSunroof` is asserted. If no car at all asserts `okSunroof`, then we have a variable flaw, as evidenced by asserting `sunroofFlaw`.

A solution to this expanded program may contain one of the predicates indicating a flaw for several reasons. One is that the particular solutions chosen for the *N* cars may simply not be the ones that demonstrate the use of each value and the absence of each potential variable. We deal with this possibility by asking the solver to minimize the number of such predicates:

```
minimize { % (excerpt)
  noLuxury, noDeluxe, noStandard,
  noBSmall, noBMedium, noBLarge, batteryFlaw,
  noAC1, noAC2, acFlaw } .
```

The solver now searches for solutions containing *N* cars that have the fewest flaws. If we limit *N* to 3, for instance, we find at least four flaws: `noLuxury`, `noConvertible`, `batteryFlaw`, `noManual`. Setting $N = 4$ produces the apparent flaws `batteryFlaw` and `noConvertible`. No matter how high we set *N*, these flaws remain.

When we try the same technique on the second example that Mittal and Falkenhainer present (we omit the second example in the interest of space), we also find both a variable flaw (can capacity) and a value flaw (the particle-physics value of the ontology variable).

It is instructive to note that only four cars are needed to cover the reachable parts of the variable domains; we might have expected that far more are needed. We can inspect these cars to verify that all reachable values are covered and that potential variables can be omitted, as in Table 2.

6. OPTIMAL CARDINALITY CONFIGURATIONS

We might often be interested in finding solutions to a set of conditional constraints that involve the fewest number of options or the largest number of options. We briefly demonstrate how such queries can be answered using our ASP model. We can use the **`minimize`** construct of *lparse* with our original formulation (before we add the numeric argument) to find a minimum solution, that is, a solution with the fewest potential variables. The requirement we add is simply as follows:

```
minimize {
  battery(bsmall; bmed; blarge),
  sunroof(sr1; sr2),
  airConditioner(ac1; ac2),
  glass(tinted; notTinted),
  opener(auto; manual) } .
```

Table 2. *A set of configurations covering all reachable values in the domains of each variable*

Package	Frame	Engine	Battery	Sunroof	AC	Glass	Opener
Standard	Sedan	elarge	blarge	sr2	ac1	Tinted	Manual
Standard	Sedan	esmall	bsmall	—	—	—	—
Deluxe	Sedan	emed	blarge	sr2	ac2	Not	Auto
Luxury	Hatch	esmall	bmed	sr1	ac2	Not	—

Using the *clasp* solver for our *lparse* model we obtain 18 optimal (minimum) solutions, including those presented in Table 3. Similarly, by using **`maximize`**, we can enumerate all 176 maximum solutions, such as those also presented in the table.

7. FMs

We now consider FMs, another form of specification that is encountered in domains such as software configuration, in which the architecture of an artifact is represented graphically. Although FMs appear quite different from CCSPs, they have very similar purposes and yield to very similar analysis. FMs are directed acyclic graphs, where nodes are called **features** and edges imply various kinds of constraints (Czarnecki & Eisenecker 2000). A **solution** is a subset of the features that satisfies all of the constraints. If a feature is present in a solution, then all the features on the path from it to the root of the tree must also be present. A feature in the tree may be marked as **mandatory**, meaning that it must be present in any solution if its parent is present; otherwise, it is **optional**. A feature may indicate that its set of children constitutes an **OR set**, meaning that if the feature is present, at least one of the children must be present. Similarly, a feature may indicate that is set of children constitutes an **XOR set**, meaning that if the feature is present, exactly one of its children must be present. Additional nontree edges indicate that if a feature is present, its successor along the edge must also be present or must not be present.

Figure 6 is based on Segura's (2008) FM for mobile telephones. Each node in the tree is a feature that might or might not be included in any model. Filled circles above features indicate that the feature is mandatory if the parent feature is included in a model. Open circles indicate optional features. Filled semicircles under a node indicate an OR set of children; if the parent is included in the model, at least one of the children must be included. Open semicircles under a node indicate an XOR set of children; if the parent is included in the model, exactly one of the children must be included. Therefore, the `Media` feature is optional, but if it is present, the `MP3` subfeature is mandatory. The `OS` feature requires that exactly one of its subfeatures, `Symbian` or `WinCE`, must be present. The `Messaging` feature requires that at least one of its subfeatures, `SMS` and `MMS`, must be present. The `Games` feature requires the presence of the `Java` feature elsewhere in the tree.

FMs are in most ways like CCSPs. Features in FMs are like variables in CCSPs. These variables have only one possible value, which we can depict as `yes`. The mandatory, optional, and edge constraints are like activity constraints. The OR and XOR constraints do not map directly to CCSPs, however.

Given these similarities, it is not surprising that representing FMs in Alloy or ASP is very much like representing CCSPs. In *lparse*, for instance, we can indicate the mandatory nature of `Settings` and the optional nature of `Media` this way, where `N` refers to the serial number distinguishing phones:

```
1 {settings(N)} 1 :- mobilePhone(N) .
0 {media(N)} 1 :- mobilePhone(N) .
```

We specify the constraint that `Settings` is implied by any child:

```
settings(N) :- 1 {os(N), java(N)} .
```

Table 3. *Sample optimum cardinality configurations*

Package	Frame	Engine	Battery	Sunroof	AC	Glass	Opener
Sample Minimum Cardinality Configurations							
Standard	Hatch	Medium	Medium	—	—	—	—
Standard	Sedan	Large	Medium	—	—	—	—
Standard	Hatch	Large	Large	—	—	—	—
Sample Maximum Cardinality Configurations							
Standard	Hatch	esmall	bmed	sr2	ac1	Tinted	Auto
Deluxe	Hatch	emed	blarge	sr2	ac2	Tinted	Manual
Standard	Hatch	elarge	blarge	sr2	ac1	Not	Manual

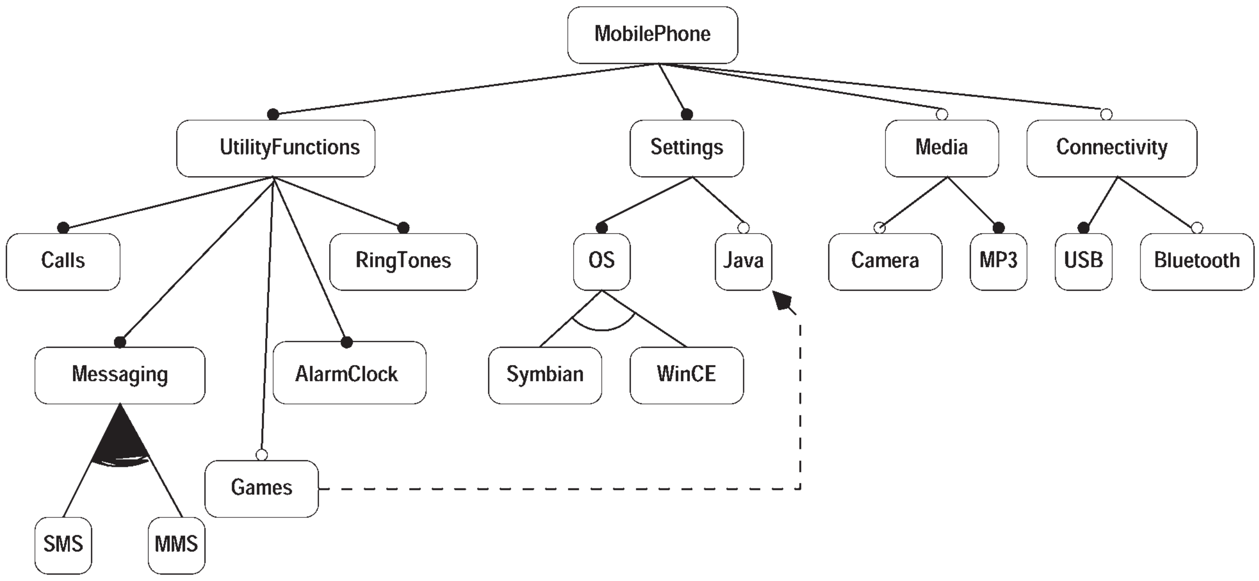

Fig. 6. A feature model for mobile phones based on Segura (2008).

We represent the OR and XOR set constraints for the children of `Messaging` and `OS`:

```
1 {sms(N), mms(N)} :- messaging(N) .
1 {symbian(N), winCE(N)} 1 :- os(N) .
```

Finally, the extra edge constraint from `Games` to `Java`:

```
java(N) :- games(N).
```

FMs are also subject to flaws. If a feature is marked as optional but exists in all solutions, the FM has a **optional-feature flaw**. If a feature is absent in all models, the FM has a **missing-feature flaw**. These flaws can result from non-tree edges. For instance, an edge from `AlarmClock` to `Symbian` in Figure 6 (Segura, 2008) would create a missing-feature flaw for `WinCE`. An edge from `AlarmClock` to `Java` would create an optional-feature flaw for `Java`.

Methods very similar to those we use in CCSPs can discover these flaws in FMs.

8. XML REPRESENTATIONS

In order to standardize how we represent CCSPs and FMs, we have designed XML Document Type Definitions (DTDs) for both, based roughly on XCSP 2.1, the DTD for CSPs (http://www.cril.univ-artois.fr/CPAI08/XCSP2_1.pdf). We have also written Perl scripts that accept instances of CCSPs and FMs obeying these DTDs, generate *lparse* renditions of the constraints, and then apply *clasp* to count or enumerate solutions, find minimum and maximum solutions, and detect flaws. In this way we can automatically generate a formal model of a configuration problem from a very natural specification.

Figure 7 shows our XML representation of the cars CCSP. The constraints typically name a variable or value as a condition and as a result. Either may be negated (as in constraint 10). The XML representation may include the logical connector and in the condition (constraint 12).

Figure 8 shows our XML representation of the phones FM, which nests `feature` nodes to mirror the picture of Figure 6.

9. DISCUSSION

Product configuration is a major industrial application domain for constraint satisfaction techniques. CCSPs and FMs have been developed to represent configuration problems in a direct and natural way. In this paper we have presented two alternative approaches to reasoning about specifications of conditional constraint sets: one approach based on well-established formal methods techniques for reasoning about software specifications, and another based on ASP. The models of the constraint specification are natural in both cases and do not require any reformulation of the original CCSP or FM. We have also shown how we could automate the testing for variable and value flaws (for CCSPs), and missing-feature and optional-feature flaws (for FMs), and that it is possible to find optimal cardinality specifications.

The DTD and Perl script are available from the authors under the GNU General Public License (http://www.gnu.org/copyleft/gpl.html). We have used this software on the fairly large "bikes" configuration (http://www.itu.dk/research/cla/externals/clib/Bike.pm), with 27 variables, some them with domains of size 14, 16, and 36. Our analyzer sets N to twice the largest domain size and tries for 10 s to minimize flaws. It then uses divide and conquer to verify each of the discovered flaws, which might be false positives due to insufficiently large N or incomplete minimization within the time limit. Each verification, however, is very fast and not subject to false positives. In the "bikes" specification, our analyzer finds

```
<instance>
 <presentation
  name="cars"
  description="Cars, from S. Mittal and B. Falkenhainer, Dynamic Constraint
   Satisfaction Problems, AAAI-90, p.  25."
 />
 <domains>
  <domain name="package" nbValues="3" values="luxury deluxe standard"/>
  <domain name="frame" nbValues="3" values="hatchback convertible sedan"/>
  <domain name="engine" nbValues="3" values="esmall emedium elarge"/>
  <domain name="battery" nbValues="3" values="bsmall bmedium blarge"
   potential="true"/>
  <domain name="sunroof" nbValues="2" values="sr1 sr2" potential="true"/>
  <domain name="aircon" nbValues="2" values="ac1 ac2" potential="true"/>
  <domain name="glass" nbValues="2" values="tinted untinted"
   potential="true"/>
  <domain name="opener" nbValues="2" values="auto manual" potential="true"/>
 </domains>
 <constraints>
  <constraint name="1" condition="luxury" result="sunroof"/>
  <constraint name="2" condition="luxury" result="aircon"/>
  <constraint name="3" condition="deluxe" result="sunroof"/>
  <constraint name="4" condition="sr2" result="opener"/>
  <constraint name="5" condition="sr1" result="aircon"/>
  <constraint name="6" condition="sunroof" result="glass"/>
  <constraint name="7" condition="engine" result="battery"/>
  <constraint name="8" condition="opener" result="sunroof"/>
  <constraint name="9" condition="glass" result="sunroof"/>
  <constraint name="10" condition="sr1" result="not opener"/>
  <constraint name="11" condition="convertible" result="not sunroof"/>
  <constraint name="12" condition="bsmall and esmall" result="not aircon"/>
  <constraint name="13" condition="standard" result="not ac2"/>
  <constraint name="14" condition="luxury" result="not ac1"/>
  <constraint name="15" condition="standard" result="not convertible"/>
  <constraint name="16" condition="auto and ac1" result="bmedium"/>
  <constraint name="17" condition="auto and ac2" result="blarge"/>
  <constraint name="18" condition="sr1 and ac2" result="not tinted"/>
 </constraints>
</instance>
```

Fig. 7. An XML representation of Mittal and Falkenhainer's (1990) car-configuration problem.

100 potential flaws in 10 s of minimization and then in another 9 s verifies that 5 are actual value flaws. Finding a solution with a given variable set to a specific value is quite fast (about 0.04 s) even in this relatively large specification; verifying a flaw takes about 0.09 s. We therefore think that the ASP approach scales well. Alloy also scales well; it is used routinely for reasoning about large complex industrial specifications (http://alloy.mit.edu/community/).

10. CONCLUSION

Our future work will study three problems.

1. We will generalize constraint-based explanation techniques so we can give advice on resolving flaws in problem specifications, thus contributing to the emerging literature on conflict detection in formal specifications (Torlak et al., 2008).
2. We will apply fault detection to configuration, so fixing the value of a variable will eliminate all newly unreachable values of other variables.
3. We will investigate how to handle nondiscrete variables, such as real ranges.

ACKNOWLEDGMENTS

Raphael Finkel's work was partially supported by the US National Science Foundation (Grant IIS-0325063). Barry O'Sullivan is funded by the Science Foundation Ireland (Grant 05/IN/I886). Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect

```
<instance>
  <presentation
    name="atomic sets"
    description="Phone example of Atomic Sets, from ??" />
  <feature name="mobilePhone">
    <feature name="UtilityFunctions" style="mandatory">
      <feature name="Calls" style="mandatory"> </feature>
      <feature name="Messaging" style="mandatory" childStyle="some">
        <feature name="SMS"> </feature>
        <feature name="MMS"> </feature>
      </feature>
      <feature name="Games" style="optional"> </feature>
      <feature name="AlarmClock" style="mandatory"> </feature>
      <feature name="RingingTones" style="mandatory"> </feature>
    </feature>
    <feature name="Settings" style="mandatory">
      <feature name="OS" style="mandatory" childStyle="one">
        <feature name="Symbian"> </feature>
        <feature name="WinCE"> </feature>
      </feature>
      <feature name="JavaSupport" style="optional"> </feature>
    </feature>
    <feature name="Media" style="optional">
      <feature name="Camera" style="optional"> </feature>
      <feature name="MP3" style="mandatory"> </feature>
    </feature>
    <feature name="Connectivity" style="optional">
      <feature name="USB" style="mandatory"> </feature>
      <feature name="Bluetooth" style="optional"> </feature>
    </feature>
  </feature>
  <constraint condition="Games" result="JavaSupport" />
  <!-- the following two constraints introduce model flaws -->
  <constraint condition="AlarmClock" result="Symbian" />
  <constraint condition="AlarmClock" result="JavaSupport" />
</instance>
```

Fig. 8. An XML representation of Segura's (2008) phone-configuration instance.

the views of the funding agencies. This work is an extension of a conference paper (Finkel & O'Sullivan, 2009).

REFERENCES

Benavides, D., Ruiz-Cortés, A., & Trinidad, P. (2005). Automated reasoning on feature models. *Proc. 17th Int. Conf. Advanced Information Systems Engineering, CAiSE 2005* (Pastor, O., & Cunha, J.F., Eds.), LNCS, Vol. 3520, pp. 491–503. New York: Springer.

Bowen, J., & Bahler, D. (1991). Conditional existence of variables in generalised constraint networks. *Proc. AAAI*, pp. 215–220.

Czarnecki, K., & Eisenecker, U. (2000). *Generative Programming: Methods, Tools, and Applications*. Reading, MA: Addison–Wesley Professional.

Finkel, R.A., & O'Sullivan, B. (2009). Reasoning about conditional constraint specifications. *Proc. ICTAI, IEEE Computer Society*, pp. 349–353.

Gebser, M., Kaufmann, B., Neumann, A., & Schaub, T. (2007). *Clasp*: a conflict-driven answer set solver. *Proc. LPNMR*, pp. 260–265.

Gelle, E., & Faltings, B. (2003). Solving mixed and conditional constraint satisfaction problems. *Constraints 8(2)*, 107–141.

Giunchiglia, E., Yu, L., & Maratea, M. (2004). Cmodels-2: SAT-based answer set programming. *Proc. AAAI*.

Hinchey, M., Jackson, M., Cousot, P., Cook, B., Bowen, J.P., & Margaria, T. (2008). Software engineering and formal methods. *Communications of the ACM 51(9)*, 54–59.

Jackson, D. (2002). Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology 11(2)*, 256–290.

Junker, U. (2006). Configuration. In *Handbook of Constraint Programming, Foundations of Artificial Intelligence* (Rossi, F., van Beek, P., Walsh, T., Eds.), pp. 837–873. New York: Elsevier.

Mailharro, D. (1998). A classification and constraint-based framework for configuration. *Artificial Intelligence for Engineering, Design, Analysis and Manufacturing 12*, 383–397.

Mittal, S., & Falkenhainer, B. (1990). Dynamic constraint satisfaction problems. *Proc. AAAI-90*, pp. 25–32.

Niemelä, I., & Simons, P. (1997). Smodels—an implementation of the stable model and well-founded semantics for normal logic programs. In *Logic Programming and Nonmonotonic Reasoning* (Dix, J., Furbach, U., & Nerode, A., Eds.), LNCS, Vol. 1265, pp. 420–429. New York: Springer.

Sabin, D., & Freuder, E.C. (1996). Configuration as composite constraint satisfaction. *Proc. Artificial Intelligence and Manufacturing. Research Planning Workshop* (Luger, G.F., Ed.), pp. 153–161. Menlo Park, CA: AAAI Press.

Sabin, D., & Weigel, R. (1998). Product configuration frameworks—a survey. *IEEE Intelligent Systems 13(4)*, 42–49.

Sabin, M., & Freuder, E.C. (1998). Detecting and resolving inconsistency and redundancy in conditional constraint satisfaction problems. *Proc. CP98 Workshop on Constraint Problem Reformulation*.

Sabin, M., & Gelle, E. (2006). Evaluation of solving models for conditional constraint satisfaction problems. *Proc. AAAI*. New York: AAAI Press.

Segura, S. (2008). Automated analysis of feature models using atomic sets. *Proc. 1st Workshop on Analyses of Software Product Lines (ASPL 2008), SPLC'08*, pp. 201–207, Limerick, Ireland.

Stumptner, M., Friedrich, G.E., & Haselböck, A. (1998). Generative constraint-based configuration of large technical systems. *Artificial Intelligence for Engineering, Design, Analysis and Manufacturing 12*, 307–320.

Torlak, E., Chang, F.S.-H., & Jackson, D. (2008). Finding minimal unsatisfiable cores of declarative specifications. In *FM* (Cuellar, J., Maibaum, T.S.E., & Sere, K., Eds.), LNCS, Vol. 5014, pp. 326–341. New York: Springer.

Raphael Finkel has been a Professor of computer science at the University of Kentucky in Lexington since 1987. He attained his PhD from Stanford University in 1976. He was associated with the first work on quad trees; k-d trees; quotient networks; and the Roscoe/Arachne, Charlotte, Yackos, and Unify operating systems. Dr. Finkel was involved in developing DIB, a package for dynamically distributing tree-structured computations on an arbitrary number of computers. His research includes tools for Unix system administration, databases, operating systems, distributed algorithms, computational morphology, Web-based homework, and ASP applications. He has published over 50 articles in refereed journals and has written two textbooks.

Barry O'Sullivan is the Associate Director of the Cork Constraint Computation Centre and a Senior Lecturer in the Department of Computer Science at University College Cork. He attained his PhD from University College Cork in 1999. His main areas of research interest are constraint programming, artificial intelligence, and optimization, with a focus on application domains such as cancer care, health, environmental sustainability, computer/network security, configuration, design, telecommunications, combinatorial auctions, and electronic commerce. Dr. Sullivan is also interested in theoretical computer science, particularly parameterized complexity and its applications.