Chapter 2
Configuration Models

Abstract This chapter provides an overview of the models used to represent the configuration problem and used for automated solving. The most common models augmenting rules and catalogs are classes, constraints, goals with restrictions, and resources. We also briefly describe Truth Maintenance Systems as a model that may be used in conjunction with these.

2.1 Rules and Catalogs

The object of computer-aided configuration is to automate as much as possible, using a computer program, such an arrangement of component instances. In order to do this, there must be a computer readable representation, a "model", of the components, the constraints, and the requirements. Were there such model, a computer could, with an algorithm that has not yet been invented, solve the configuration problem of arranging the letters of the alphabet into this text of this book. A model is necessary, but not sufficient, for automatic configuration problem solving.

One dimension of the way that configuration technologies differ is in the type of model. That is, when we describe a configuration problem solving technology, it is useful to distinguish it from others by its different features. One of the most important ways to distinguish configuration technology is by the kind of model a technology employs.

Any model of a configuration problem must contain at least the parts to be arranged as elements. This is typically a database, or catalog, of components to be selected. If we have only a finite set of actual Lego® blocks to assemble, our model consists of just these parts (types of blocks).

If we model lego block types so that a computer can reason about how to assemble together instances of them, then this is a component model that has at least two levels: a set of types and as many instances of each type as we need to configure the artifact. If we are given user requirements that refer only to the aggregate properties of such

C. J. Petrie, *Automated Configuration Problem Solving*, SpringerBriefs in Computer Science, DOI: 10.1007/978-1-4614-4532-6_2, © The Author(s) 2012

a configuration (e.g., size and shape of the boat), such a model suffices for automatic configuration with some algorithms.

In this case, it may suffice to use only rules to describe our lego blocks. We might say, for example, that all "short blocks" have three pegs/holes and all "long blocks" have five pegs/holes, and all blocks have a width of 2 pegs. Then we could write rules about how to combine such blocks: e.g., to attach one block to another, there should be at least one peg on one block and one hole on the other that are free to be used for the attachment. The block descriptions would be a sufficient model for some configuration engine to solve a given problem. An additional simple model would be to create a catalog or database of available parts.

2.2 Class Hierarchies

For other than very simple problems, a completely rule-based approach with a parts catalog is inadequate, if only for reasons of programming complexity. The user requirements may need to be stated in terms of larger components and/or aggregate functional requirements. For example, we may want to configure an automobile that can go 120 mph. Because we want it to go that fast, we know that the properties of the sub-components, for example the wheels and engine, will need to provide and support such speed. Such requirements tend to be of two kinds. One is a requirement for subcomponents: e.g., a computer should have a CPU, memory, and an I/O bus. Another is for functionality or capacity: e.g., the computer should have at least 1 TB of storage. Both such requirements will lead to inclusion of new subcomponents in the configuration.

In such cases a more complicated model using class hierarchies is useful so that we can easily compute the properties of the design components, possibly through inheritance. Other relations may also provide additional guidance for automated problem-solving. The use of such a model is often called "model-based" configuration but it is so common that almost all approaches that perform complex configuration have some kind of class model, even if embedded in rules.

In a class hierarchy, subclasses descend from superclasses. Each element of such a decension tree is a class or an instance of a class. Each subclass will have at least one superclass. This is commonly called a "is-a" relationship. Subclasses are specializations of their superclasses. Members of any subclass inherit the attributes, or slots, of a superclass from which it is descended. They have further attributes make the subclass a specialization of the superclass. So "Jaguar" is a subclass of "Automobile" that has more specific attributes than does a general automobile. The component "engine" may have specializations of "gasoline engine" and "diesel engine".

Complete class hierarchies must include a class layer of components which have no specializations or sub-components. These are often called "primitive", or "leaf", components. A configuration will consist of instances of these primitive components. For example, the class "BMW Model 531e" may have no further subclasses.

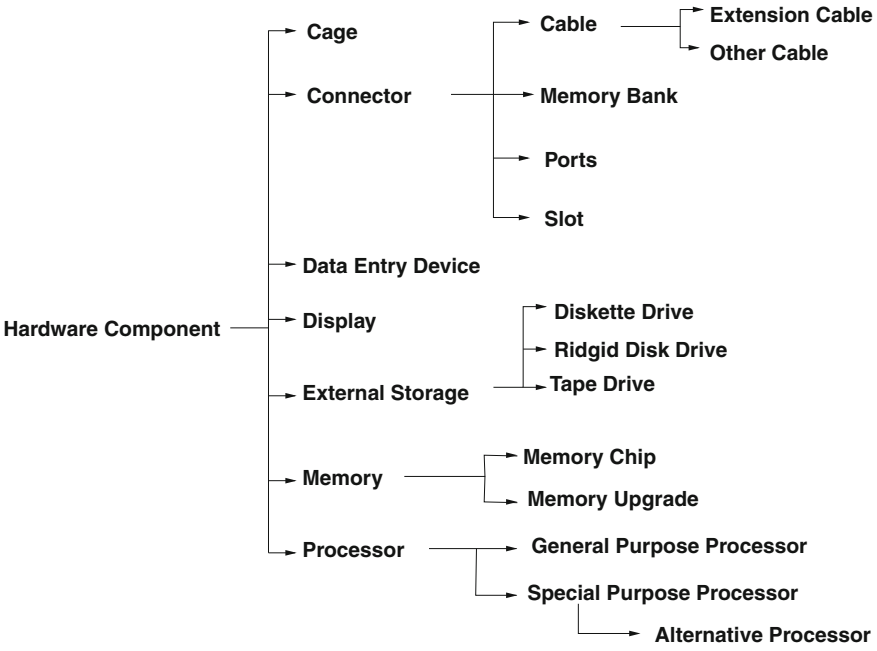

Fig. 2.1 COSSACK class hierarchy

The design configuration would be complete with such classes, but a model could contain class instances, such as "VIN930bo193x0222".

It is difficult to track down the earliest instances of such class hierarchies, but certainly one of the earliest famous cases is that of the general (domain independent) expert system shell EMYCIN [68] that included a hierarchical structure called a "context tree". Each object in this structure was of a particular type, associated with certain parameters. Parameters were inherited from parents (superclasses) and there could be instances of each type.

A good example of such a class hierarchy is shown in Fig. 2.1 which is a simplification of the hierarchy from COSSACK [20]. The top level superclass is "Hardware-Component". Subclasses of this top level component class include "Connector" and "Cage". Connectors include the subclasses "Slot", "Ports", and "Cable". "Cage", which is a type of component that may contain other components, is a primitive, or leaf, component as it has no subclasses.

Even some of the earliest rule-based systems used some simple class hierarchies. For example, Fig. 2.2 shows a simplified example of R1/XCON component descriptions [39].

Class, or "frame", systems are very specific object-oriented technologies that predated the mainstream use of object-oriented programming (OOP) [60] but inspired it. In frame systems, the objects are data types that inherit from one another. In OOP,

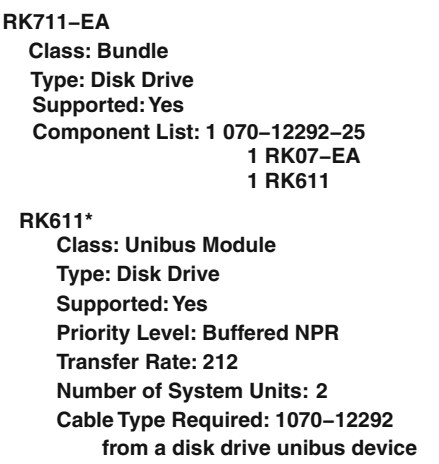

Fig. 2.2 R1/XCON class hierarchy

the objects are software components with datatypes with behaviors, which can be inherited. However, they are two distinct technologies that should not be confused.

As shown in [2] and [69], frame class hierarchies and the properties of the classes and their inheritance could be represented in logical rules. As these became understood as static structures, it was understood that it was more efficient to use static specialized frame representation systems such as those used in Knowledge Craft, KEE [29], and the MCC "Proteus" [46] framework that combined forward and backward-chaining rules, truth maintenance, and frames. Such frame/class systems had attributes, often called "slots", associated with each class (not to be confused with the common domain-specific class "slot" referring to a specific location in the artifact where a component may be inserted).

Later systems use other types of classification subsumption systems that have equivalent but perhaps more efficient mechanisms, such as the use of description logic [41] in PLAKON [10, 27].

Whether done in rules or more efficient frame/slot systems or possibly even more efficient description logic, constraints could be placed upon the values of attributes of each class, such as cardinality. For instance, an "automobile" might be a class with the attribute "wheels" and the cardinality of this attribute might be restricted to the number four. Such class systems are frequently also called "ontologies" as the they are not simple taxonomies but more complex models that constrain the use of the elements in the models. (Ontologies that use various forms of computational logic [22] are used today to define formal semantics for various terms on the Wold Wide Web.)

In addition to the "is-a" relation, classes may have arbitrary relations with respect to each other. One that is very useful for configurations is the "part-of", sometimes called "child-of" or, inversely, "has-part", relation that describes sub-components [2] that are necessarily a sub-component of a super-component. This is an alternative to using the attributes of a class to constrain the class instantiations.

For example, instead of defining the class “automobile” as having an attribute of “wheels”, this a could alternatively be defined as another class with a “part-of” relationship to automobile. To elaborate the example, the class “automobile” might be defined to always have a “has-part”/“part-of” relationship to the classes “engine”, “chassis”, and “wheel”. In addition, constraints are often stated that an automobile should contain one each of these first two classes and four of the third. Other types of common relationships are “connects” and “contains”. Similarly, an “engine” may require the sub-component “piston”. A valid configuration respecting such a model would always result in car designs that have one engine, one chassis, and four wheels, and the engine would have at least one piston.

Any given model may have any finite number of such pre-defined relationships in order to guide problem solving. The semantics of the relationships and constraints is ultimately defined by the problem-solving algorithm that uses them. Some simple configuration systems force such sub-components selection [21].

In general, a configuration may not necessarily include component instances of all sub-components of a component. More general configuration systems may represent sub-components as optional parts. For example, a “computer” may have a sub-component of “optical disk reader” but this component is optional. Or it may be that some kind of optical disk reader is required but this might be either a “CDROM” or a “DVD” reader. If no subclass was defined, then either one, but at least one, must be part of the configuration. In this case the classes would not represent mandatory sub-components but there would be a rule that at least one instance of one of these subclasses would be in the configuration. For example, we don’t know in the beginning of the configuration problem whether this type of computer will have a CDROM or a DVD but we know it will have one or the other.

In addition, some configuration systems use a functional hierarchy that uses the is-a hierarchy to inherit functional properties, which has some advantages in satisfying functional requirements and constraints [4, 21]. This is similar to OOP in that artifact functions and software behaviors are at least analogous.

2.3 Constraints

A constraint is a very general and useful formalism. It is often expressed as a rule about the combinations of the assignments of values to variables. A common example is map coloring: “any two adjacent countries must have different colors”.

A common way to express this is as a triple: “(Color Germany Red)” would mean that he color of the map element “Germany” has been assigned the “color” “red”. However, for constraint representation, it is convenient to use a single variable name, such as “Cgerm”. Then color of any country on the map is expressed as the value of a variable. So, in addition to “Cgerm”, the color variable of Austria might be called “Caus”, and that of Switzerland might be Cswiss. We may have only the color values of “blue”, “red”, “green”, “white”, and “black” to assign to these country color variables. If we assign “blue” to “Cgerm”, then “Caus” can only be one of the

other three colors because of the constraint. Assigning one, say "red", to "Caus", means that "Cswiss" must be either "white" or "black".

To get around the awkward variable names, sometimes variables are abbreviated with a particular syntax, such as "Germany:Color". Then our constraint might be expressed as

$$\neg \wedge (?X:Color \ = ?Y:Color)(Adjacent?X?Y).$$

This relates to our previous discussion of classes because class have attributes, also called "properties". "Color:Automobile" would be the variable relating to the class Automobile. An assignment this high in the class hierarchy is typically a default that propagates downward. For example, "(Color Automobile Black)" may be used to mean that all automobiles (in this model) are colored black.

Constraints are conditions over the assignment of values to the properties of components in a configuration that allow only some to be valid. The constraints may be very specific to the values assigned to a slot on a frame or more general in how components can be combined. A simple example of a constraint is that an automobile may have no more than four wheels.

Constraints are a common part of models used to represent and solve a configuration problem. In some systems, only constraints are used. That is, constraints may only be part of a class model, or they may be the only model used. We discuss the latter in Sect. 3.2.

The constraints associated with configuration design problems often include constraints on how instances of one component type are connected to another, often with an instance of a third kind of component called a "connector", where some constraints determine that particular components can be connected only at special points called "terminals", which may be "slots" (distinct from frame slots) and/or "ports" [1, 20, 21]. Typically a "slot" is a terminal into which some component is inserted rather than just connected to, as with a "port". Typically, the restriction of port and slot connection points also includes a new requirement for a particular class of connector, such as a USB cable.

All constraints, including connection constraints, may be associated with components at any level of the specialization hierarchy tree. All components lower in the tree inherit these constraints from components higher in the tree. For example, every "disk drive" must be connected to a "system bus" at a "bus interface port". So specialization disk drive "160 GB-drive" component must be connected at a specialization of such a port by some specialization of such a bus, which is a specialization of "connector".

A related common type of constraint associated with configuration design problems includes spatial constraints. Some components must have a particular spatial relationship with others. This may involve being nearby or being physically inserted into each other, such as a board into a slot. Typically there are restrictions on how many and which components can be inserted into the slots of another component.

Some of these constraints represent physical constraints (no more than six power adapters of any kind may fit on the power strip and no more than three adapters

larger than one inch may fit) and some have to do with functional requirements such as power (the total power requirements of all power adapters on the strip may not exceed 500 W).

Constraints may be represented as rules apart from a class system, and/or as constraints associated with the component classes and/or their slots. In either case, they are logically equivalent and express relations constraining the valid configuration.

2.4 Constraints as Goals and Restrictions

In some systems, all conditions on components are called "constraints". The solver simply finds value assignments to the variables that satisfy all of the constraints, possibly with the aid of class models and other types of knowledge. However, it is often useful to distinguish between different kinds of constraints. Such a distinction is particularly useful for Generate and Test problem solving methods, discussed in Sect. 3.3. These methods will typically use a constraint and class model discussed above, but with further distinctions.

Some constraints are restrictions on how components existing within a given configuration may be combined: these are compatibility constraints. For instance, such a constraint restriction might be that a convertible may not have a hard top. If during the configuration synthesis of a convertible car, a hard top is added to the configuration, then this constraint is violated. In order to fix a constraint violation, we must either remove the hard top from the configuration or remove the selection that the car is to be a convertible. Connection, physical, and spatial constraints are other types of restrictions.

Some constraints may restrict the aggregate of resource consumption. A typical type of restriction constraint is a budget of some kind: the aggregate of components added to the configuration cost too much money, weight, or some other budgeted quantity, including perhaps time to build. It is often not possible to fix such a violation by simply adding something else to the configuration. A notable exception are resources, such as money and electrical power. For example, it may be possible to add an additional power supply. If this is not seen as replacing one power supply with another, then the available power is just a constraint and so the constraint itself is changed (or this is represented as a compatibility constraint). So the solution is either to change an earlier variable assignment or to change the constraints.

Other types of general constraints are requirements, or requests, that some additional components or statements about the design should be added to the configuration, if these requests are not already fulfilled. If we have selected the car to be a luxury touring car, then there may be an added requirement that a multimedia package be added to the car. This is a request that can be satisfied by adding such a package to the configuration if we have not already done so.

So far, we have discussed restriction and requirement constraints, Both can be handled by constraint solving systems [27]. However, there is a difference between the them that can be exploited by a problem-solving system. As discussed above,

typically the way that constraint violations of the first type is solved is to change a previous selection of variable assignment. Such constraints are not violated unless such selections are made. However, the requirement constraint may be initially violated until some selection is made. The same is true for resource requirements, such as a requirement for disk capacity in a computer. The more general notion of such requirements is that of a *goal* [50, 71].

While sometimes the choice between saying a condition is either a goal or a restriction is a representation decision, it is often useful to distinguish such requirement-type constraints that can be satisfied by making design decision from restriction-type constraints that are only violated when such decisions are made. One reason is that there are often many more possible restrictions to be considered than are tractable and proceeding by satisfying necessary goals until some restriction is violated often reduces the constraint problem space. In addition, the goal representation facilitates configuration task decomposition as it explicitly labels the expression as possibly leading to decomposition, distinguishing it from restrictions. A goal representation also ensures that there is a valid reason for each of the component assignments, thus providing an explanation and eliminating unneeded assignments that may have occurred as a by-product in pure constraint satisfaction.

The component instances in a correct, or valid, configuration have properties that collectively satisfy the top-level goals of the user do not violate the constraint restrictions that may be associated with the component properties. It may not be possible to satisfy all of the requirements and restrictions for a given problem, in which case the configuration will still consist of a set of component instances but will not be a correct configuration. Still, it may be the best that can be done because the problem is over-constrained. Also, it may be that constraint violations should be deferred for a while during problem solving but ultimately resolved producing a correct configuration [50, 71].

2.5 Resources

It is common in many configuration systems to represent some component properties as well as some user requirements as resources. Indeed, some configuration systems use only resources [25] for configuration problem solving.

Resources are commodities that may be both provided and consumed by components, and may be required by the user. An individual component may supply a unique resource: i.e., a particular CPU provides 10 GHz serial processing speed. Or it may supply a resource that aggregates. If the computer needs a total of 500 GB of disk space, this may be achieved by five 100 GB disk drives. If the two disk drives plus the CPU consume 100 W of power, then this power may be supplied by either a single power supply of 100 W or by configuring two 50 W power supplies that work together.

In such systems, there is typically no distinction made between requirements and restrictions: there are only resources needed and consumed. Such resources are often

represented as properties of the individual components. The property of a disk drive may be that it consumes 50 W. The property of a power supply may be that it supplies 50 W. Resources may concern component aggregate properties. A user resource goal, which is really a restriction, may be that the total weight of the components be less than 2 lbs. A resource requirement-type constraint may be that if the total heat output is more than 30 W, then a fan must be configured. Resource-oriented systems treat all such resource constraints similarly. Indeed, all requirements and constraints are considered as resource provision or consumption that needs to be balanced. This is a unique view on configuration and though successful, has not been applied much since the original work [25]. Typically this resource computation is combined with other types of problem-solving [27].

2.6 Truth Maintenance

While they are only ever used in conjunction with other models and with specific reasoning systems, Truth Maintenance Systems [12, 16] are an important model for configurations systems. Essentially these systems provide a background support to ensure the consistency among some set of beliefs. The model is complex but the essential idea is that the reason for belief in some assertion should be supported by belief in some set of other assertions, or lack of belief in them.

The simplest use of a TMS is to record the results of firing a rule. For instance, if the rule is that

$$\wedge(?X : Daytype{=}Weekend)\neg(?X : Weather{=}rain) \Rightarrow (?X : Picnic = Yes)$$

then we can plan a picnic for a day on the weekend as long as we don't believe it rains on that day. Upon firing this rule for a particular day, say "Saturday", a TMS will add a new justification node for having a picnic on Saturday to a possibly already existing network of justifications. Figure 2.3 shows a simple example justification that might result from such a rule firing, with the belief that Saturday is a weekday somehow also justified. The *IN* and *OUT* node labels refer to whether a node has a valid justification: the former if so and the latter if not. A justification is valid if all of its supports marked with + are *IN* and all of its supports marked with − are *OUT*. Any node may have multiple justifications. See [50] for a further explanation.

The important point is that a TMS will automatically retract the belief in planning for a picnic on a day as soon as it is believed that it will rain that day. This functionality can be of use in tracking the validity of configuration choices.

However, the use of either the original TMS or the ATMS has long been understood to be problematic [48], because, if one simply ties a rule-firing to the production of a TMS justification structure, the propagation of change in the TMS network can be difficult to understand and predict. For instance, a top-level choice in the customization of an automobile as a sports car might lead to many choices, such as the choice of high-profile tires. If the top-level choice is revised, then all such

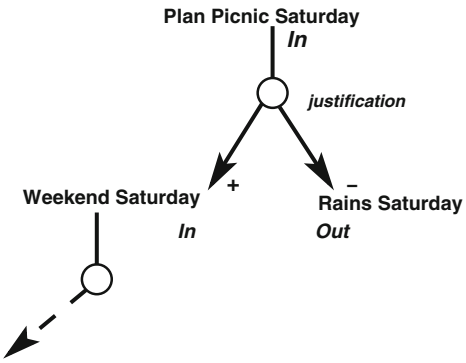

Fig. 2.3 A simple TMS justification network

further choices including the choice of tires will be automatically revised as well, even though the tires may actually still be desired in the new sedan configuration. Given a top-level rule firing that is changed, all subsequent rule firings that lead from the consequent of that rule will be changed whether or not a domain model requires such change. In the worst case, work should be repeated and in the worse, interactive configuration will be confusing. A use of TMS in configuration is described in [50] and discussed in Sect. 4.3.

http://www.springer.com/978-1-4614-4531-9

Automated Configuration Problem Solving

Petrie, C.J.

2012, VII, 38 p. 4 illus., Softcover
ISBN: 978-1-4614-4531-9