12

Configuration

Deborah L. McGuinness

Abstract

Description logics are used to solve a wide variety of problems with configuration applications being some of the largest and longest-lived. There is concrete, commercial evidence that shows that description logic-based configurators have been successfully fielded for over a decade. Additionally, it appears that configuration applications have a number of characteristics that make them well-suited to description logic-based solutions. This chapter will introduce the problem of configuration, describe some requirements of configuration applications that make them candidates for description logic-based solutions, show examples of these requirements in a configuration example, and introduce the largest and longest lived family of description logic-based configurators.

12.1 Introduction

In order to solve a configuration problem, a configurator (human or machine) must find a set of components that fit together to solve the problem specification. Typically, that means the answer will be a parts list that contains a set of components that work together and that the system comprised of the components meets the specification. This task can be relatively simple, such as choosing stereo components in order to create a home stereo system. The problem can also be extremely complex such as choosing the thousands of components that must work together in order to build complicated telecommunications equipment such as cross-connect devices or switches.

One important factor that makes configuration challenging is that making a choice for one component typically generates constraints on other components as well. For example, if a customer chooses a receiver that only supports up to four speakers, then she may not conveniently support a surround sound system with a subwoofer (since this would require more than four speakers).

Configuration continues to have strong interest in the academic and commercial communities. It has been a prominent area in artificial intelligence at least since the R1/XCON [McDermott, 1982] work on configuring computer systems. Since then, many configuration systems have been built in domains including communication networks, trucks, cars, operating systems, buildings, furniture layout, and even wine properties to match a meal description. Today, there are active mailing lists, workshops and conferences (such as the configuration workshops at IJCAI 2001 [Soininen *et al.*, 2001], AAAI'99 [Faltings *et al.*, 1999], and the Fall Symposium Workshop on Configuration [Faltings and Freuder, 1996]), special issues of journals (such as IEEE Intelligent Systems [Faltings and Freuder, 1998] and Artificial Intelligence for Engineering Design, Analysis and Manufacturing [Darr *et al.*, 1998]), and research groups at a number of universities and companies. Approaches include constraints, expert systems, model-based reasoning, and case-based reasoning as well as description logics.

Configuration is an important and growing commercial concern. There are a number of companies dedicated to configuration such as Trilogy, Calico, etc. Other companies in broader markets such as the enterprise integration software companies, Baan and SAP, have a major emphasis in configuration. Companies that sell complicated products, such as computers, are providing their own configurators (e.g., the Dell personal computer online configurators). There are spin off companies of general configuration companies that are aiming at particular domain areas, such as PCOrder (a spinoff of Trilogy focusing on personal computer configuration). There are also some domain-oriented companies that include configuration as a major component such as CarsDirect's configuration of United States consumer car orders.

Although the commercial configuration market may appear to be a recent event since it has been exploding recently, it does have at least a decade of history. Trilogy, for example, one of the earlier companies focusing primarily on configuration, was founded in 1989. Forrester research reports that the configuration market was valued at eight billion dollars in 1997 and it predicts that the market will grow to 327 billion in 2002. Configuration is also seen as important by companies not originally classifying themselves as "configuration companies". In a study of fifty eCommerce executives from top firms in the business to business and business to consumer space, Forrester Research found that search and configurators were considered the two tools most critical for customer support [Koetzle *et al.*, 2001].

The description logic community has been addressing configuration needs for over a decade as well. Owsnicki-Klewe [1988] presented a view of configuration as a consistency maintenance task for description logics and AT&T independently began work in 1988 on its family of configurators for telecommunications equipment [Wright *et al.*, 1993; McGuinness *et al.*, 1995; McGuinness and Wright, 1998b;

1998a]. Similarly Ford Motor Company has had a description logic-based configurator [Rychtyckyj, 1996] in the field for over 10 years. Others in the description logic area have explored description logics for configuration as well, e.g., [Buchheit *et al.*, 1994c; Kessel *et al.*, 1995].

12.2 Configuration description and requirements

In this chapter, we will be considering large scale configuration problems. If one only has a small number of constraints to satisfy and a small number of possible component choices, then any somewhat reasonable solution will work. If however, the final product is complicated and there are thousands of choices and constraints, then there is more need for a well suited solution. We will consider the generic configuration problem where there is a complex artifact being assembled from components. Potentially the components have subcomponents, thus the artifact may be modular or hierarchical in nature. Also, each of the components typically has a number of properties, such as power restrictions, connections to other components, etc., thus components may be tightly interconnected. If one looks at modern configuration descriptions [Fleischanderl *et al.*, 1998; Juengst and Heinrich, 1998], one can see only large, interconnected, tightly constrained, complex systems.

The input description for the configuration problems we will consider will be a specification of a complex, probably highly interconnected system. The input should be able to be input incrementally by a user as well as being able to be uploaded from sales programs. The input specification may be:

- incomplete
- ambiguous
- incrementally evolving
- granular to different levels of specificity
- inconsistent
- entered in any arbitrary order
- interconnected
- nested with complex structure

The output for the system, in its simplest form, will be some kind of parts list. The parts list may be organized hierarchically so that there is a parts list of high level components (such as bays in switching systems or speaker sets in home theatre systems) as well as a detailed parts list of the individual components. In this chapter, we will only address configuration and not the related area of parts layout.

The output of the system should be:

- correct

- complete
- consistent (with respect to other parts, preferences, pre-existing components in the customer's environment)
- modifiable
- understandable / explainable
- capable of being queried
- interconnected and interoperable with related data

The configurator needs to accept the problem input along with any previously entered domain information concerning valid configurations. It must then check the constraints it has (calculating the constraints that are implicit in the input data from the input and background information) in order to start building a parts list. It may find that a complete and correct parts list may not be built from the given input. In actuality, it is common for the problem specification to be either overconstrained (i.e., contain a contradiction such as "I want a pair of speakers that is of the highest quality available yet I do not want to pay more than fifty dollars for them") or underconstrained (i.e., "I want to buy a high quality stereo system"). In the first case, the configurator needs to identify the source of the conflicting information and determine (probably along with user input) which conflicting constraint(s) to relax. In the second case, the configurator needs either to solicit more specific information from the user, or to generate a list of possible configurations, or both. If the configurator makes arbitrary choices for the user (e.g., it chooses some receiver for the stereo system yet there were many possible choices), then it needs to make it possible for the user to change the arbitrary choices and also to find out which choices were arbitrary and which choices were mandated by constraints. Additionally it needs to let the user input partial additional input that would further constrain the choices.

The configurator also needs to accept information from multiple data sources. There will be a number of databases with which a configurator may need to interact. Typically, there will be databases of parts and prices, other databases of parts and availability, and possibly many other databases with user information or just information about different product families. It is likely that information (such as pricing and availability) will change frequently. Also, there will be information concerning what parts are compatible together and how the choice of one part constrains the choices of other parts. These might be considered the configuration rules. These rules might not change on a frequent basis, however modifications are typically necessary. The rules may come from multiple sources as well. They may need to be imported from many different source languages and they may need to be input by people who have no training in computer science, let alone knowledge representation systems.

Finally, the system may be long-lived and thus require support and maintenance. It may be necessary to staff a help desk to help users of the system. The customer service representatives may know very little about any one individual product for which they are answering questions (because they are supporting a large number of products). The technical staff maintaining the individual configurator may not include people who originally built the system, and over time, it may not even include people who know much about the product (although they may be quite capable of researching the product if necessary). Also, the technical staff may need to generate new configurators for updated or similar products.

We might summarize the requirements from the input, output, and core configurator requirements starting from the requirements presented in one configurator family of applications [McGuinness and Wright, 1998b] and augmenting them slightly here. A solution methodology should have the following properties:

- object-oriented modeling;
- rule representation, organization, and triggering;
- active inference and knowledge completion;
- explanation, product training, and help desk support;
- ability to handle incrementally evolving specifications;
- extensible schemas;
- reasoning mechanisms that handle incomplete or ambiguous information;
- inconsistency detection, error handling, and retraction;
- modularity;
- maintainability.

This list of needs represents those in many complicated reasoning tasks. Although we could argue that this general architecture and approach is more broadly applicable, we will limit our discussion to configuration applications. In the next set of subsections, we will describe each of these needs with respect to the task of configuring a stereo system (based on the configurator demo by AT&T [McGuinness *et al.*, 1995; 1998] and mention how the description logic-based solution met the need. When useful or necessary, we will mention how the need was addressed in the larger PROSE configurator family.

In the stereo configuration application, the goal was to require the user to enter a small number of constraints concerning the end system and generate a complete, correct, and consistent parts list. The system would accept a large set of constraints as input as well, however the goal was to reduce the user's task and thus require minimal input. The system used the user input along with its extensive domain knowledge and parts information to determine if the user's input specification was consistent. It used the underlying theorem prover within the description logic system to compute the deductive closure of the input and generated a more complete input description. User input was solicited on the system quality (high, medium, or low with associated price ranges) and the typical use (audio only, home theater only, or combination), and then the application deduced applicable consequences. This typically generated descriptions for 6–20 subcomponents which restrict properties such as price range, television diagonal, power rating, etc. A user might then inspect any of the individual components possibly adding further requirements to it which may, in turn, cause further constraints to appear on other components of the system. Also, a user may ask the system to "complete" the configuration task (even if the user specification was incomplete), completely specifying each component so that a parts list is generated and an order may be completed. An online demonstration of the web configurator application is available at Vassar (`http://taylor.cs.vassar.edu/stereo-demo/`) and a number of examples are available in the extended online version of the IJCAI paper [McGuinness *et al.*, 1995] available at: `http://www.research.att.com/sw/tools/classic/tm/ijcai-95-with-scenario.html`.

This application is convenient for illustrating our points since it is small and in a broadly understandable domain. It is potentially more interesting than some simple pedagogical examples since it was developed as an application that had representation and reasoning requirements that were isomorphic to the needs observed in the Prose family [Wright *et al.*, 1993; McGuinness and Wright, 1998b] of configurators. The examples in this paper can be seen in more detail in [McGuinness *et al.*, 1995; 1998]

12.2.1 Object-oriented modeling

A system that is being configured may be viewed as a structured object composed of smaller objects. Even our simple example domain of stereo equipment presents a natural hierarchy of concept descriptions and instances that have a number of properties. We have a top level node like ElectricalThing and then have subclasses of that node such as HomeTheatreSystem and StereoOrVideoComponent. Further, subclasses of StereoOrVideoEquipment might include Receiver, Speaker, and Television. Any particular term may have properties associated with it. For example, a Television might have a property called diagonal (that must be filled with a positive integer), another called price (that must be filled with a monetary value), a repairHistory (that must be filled with one of the following values: {BAD, OK, GOOD}), a manufacturer (that must be filled with a company), and a height, width, and depth (all of which must be filled with a positive number). All of the properties might have cardinality requirements on them. For example, there must be at least one manufacturer (although possibly more than one manufacturer), there must be exactly one filler for the diagonal role, etc.

In the simple examples so far, we have seen a need for number (cardinality) restrictions, value restrictions (choosing the type of a filler for a role), roles, and class hierarchies. Further we should note in the description that the objects are compositional. The value restriction on the manufacturer role is naturally determined to be a company. Companies themselves might have further properties like headquarter locations, CEOs, etc. A user might subsequently want to choose speakers made by companies in the United States and televisions made by companies headquartered in Japan.

It is argued more extensively elsewhere [McGuinness and Wright, 1998a] and in this book in Chapter 10 that description logics are convenient modeling tools for such objects. We can show a simple example of this diagrammatically where a HomeTheatreSystem inherits a price role with a value restriction of MonetaryUnit. We might also have a particular HomeTheatreSystem named MY-HTS that is the system we will be building through the example. It will also have a price role with some unknown value at the moment. We might also have a subclass of HomeTheatreSystem called HighQualSystem. In our simple example, this might be defined simply as a home theatre system that costs at least 6000 dollars. In a description logic system, once MY-HTS contains either a price that is over 6000, or contains a partial description such as "a minimum price of 8000 dollars" that restricts the price to be greater than 6000, then it can be recognized to be an instance of a HighQualSystem. This kind of automatic recognition and organization of terms based on their definitions is a convenience for organizing and maintaining partial descriptions and is arguably one reason that description logics are thought to be particularly useful for modeling and maintenance of applications that require object-oriented models.

12.2.2 Rule representation

A knowledge base that contains information about active deductions will contain some sort of rules. Typical large configuration systems will contain many rules. Also, these rules may change frequently. It is reported that 40% of the rules in R1 changed yearly. Thus, support for modeling, organizing, and later, maintaining the rules will be important in large configuration systems. A simple rule may take the form of "If something is an A, then it is a B". For example, if something is a HighQualSystem, then its television is a HighQualTelevision (which has a minimum price and diagonal value), its speakers are HighQualSpeakers (which have minimum price restrictions), etc. In fact, in our stereo demo, there are dozens of rules that fire once a system is determined to be a HighQualSystem. If the minimum price restriction were ever removed from the specification requirement, we would want the results of those rules retracted automatically (unless the same results could be deduced in other ways as well).

A description logic-based system can support modeling of rules described above in a hierarchical fashion. Rules can be associated at what ever level of the hierarchy is appropriate. Thus, we might associate minimum price and diagonal for televisions at the level of a HighQualSystem and we might associate repair-history restrictions with another concept such as HighReliabilitySystem. If we just wanted to have this kind of simple rule encoding, one would not have needed to use a separate mechanism. If one has an encoding scheme that includes negation and disjunction (or some other way of encoding an "if-then" rule), as do most of the modern description logic languages, then one does not need to introduce a separate rule notion. For example, one might encode a simple if-then relationship such as `(or (not HighReliabilitySystem) GoodRepairHistory)`. This states that either something is not a high reliability system or it has a good repair history, which is typically viewed as equivalent to "if something is a high reliability system, then it has a good repair history".

The description logic that this example was encoded in (Classic [Borgida *et al.*, 1989; Brachman *et al.*, 1991; Patel-Schneider *et al.*, 1991; McGuinness and Patel-Schneider, 1998]) had a rather limited set of constructors and also had the simple rules introduced above and also more sophisticated rules such as those which compute role values based on context. In some configuration applications of this description logic, the more sophisticated rules in combination with other constructors have encoded expressive rule-based reasoning, and in fact many of the rules in those configuration system required Classic's more sophisticated rule representation system. The examples we have seen in this chapter only use a simple form of if-then rules. For a more detailed discussion of how powerful these rules can be in practice, see [Borgida *et al.*, 1996].

Description logics are not required of course in order to capture rule representation and reasoning, this example simply shows that they can be a convenient technique for capturing rules and reasoning with them.

12.2.3 Active inference

Description logics deduce logical consequences of information and are thus said to provide active inference. In fact, one of the typical patterns of inference observed in many description logic-based configuration systems includes

- Asserting new information about an existing term
- Recognizing that the updated term is an instance of a class
- Firing a rule on the term that is associated with the class
- Propagating information from the updated term to related terms

For example, lets consider MY-HTS again. Let it have a hasTelevision slot filled with a particular television TV-11. Once it is asserted that the user is willing to pay more than 8000 dollars for this system, it is recognized to be an instance of the HighQualSystem. The rules associated with that concept fire and now it becomes an instance of something that has a television diagonal minimum of 50 inches (or possibly a high definition television with a smaller diagonal) and a television price of a minimum of 1000 dollars. These restrictions are propagated onto TV-11.

This kind of deduction chain comprises over 50% of the inferences that are done in the stereo configurator example. In this manner, users only need to specify a small number of restrictions on their system and they can have a large number of deductions performed for them.

It should be noted that this particular example configurator was built on a description logic that did not contain default reasoning. Some description logics have been expanded to include default reasoning (i.e., if it is not known to be otherwise, use the default rule) [Padgham and Zhang, 1993; Baader and Hollunder, 1995a; Quantz and Royer, 1992]. For example, if a manufacturer has not been specified for a television, use Sony as the manufacturer. If the underlying formalism had a default representation, this would have been used.

As the demonstration system was encoded, the stereo configurator used two sets of concepts on which to hang rules - a concept for all provably correct rules (such as power compatibility) and another concept for the default rules, called a “guidance” concept (for more subjective rules such as minimum prices). The deployed configurators on which this system was based actually used defaults as completion—at a particular point in the specification input process, if information is unknown, then “complete” it using the “default” or subjective rules [McGuinness and Wright, 1998b]. This provided one very simple method of implementing a kind of “default” as completion that can be viewed as one of the simplest forms of default reasoning.

12.2.4 Explanation

Customer help desk staff need to be able to help users understand potentially everything about a configuration specification and the final parts list. In fact, the Prose family of configurators faced extinction had it not been able to respond with a full explanation capability. It was evident that consumers needed to be able to find out why some particular part was in their final system, why it had the particular value restrictions it did, what the possible alternatives were, and from what portion of the specification this information had been derived. In this simple example, a customer might want to find out why the television in her final system costs over 1000 dollars or why it has a particular minimum diagonal requirement. The explanation would be that a high quality system was requested and high quality systems include a suggested minimum diagonal size and a minimum price on their television components.

The demonstration system allows customers to point to particular components and ask questions about everything that has been deduced about them. It also anticipated the most common explanation questions that users asked and provided pull down menu items that were dynamically generated based on the item a user was pointing to to generate explanation questions that a user could just click on to ask quickly. An extensive explanation foundation was designed for the underlying description logic-based system in order to support that [McGuinness, 1996; McGuinness and Borgida, 1995]. The explanation system provides a proof theoretic foundation for explaining any deduction in terms of proof rules and arguments. It also provides an automatic followup capability that generates the questions that would lead to this inference being deducible. The followup question generation was found to be needed since user studies showed that users wanted fairly simple explanations along with the capability to ask followup questions. Further studies found that users appreciated help in generating syntactically correct followup questions that made sense given the previous question that was just asked. The followup questions were automatically generated from the model-theoretic form of the explanation.

The basic explanation structure was originally done for a normalize-compare description logic-based system but has since been used as the foundation for a tableaux-based description logic [Borgida *et al.*, 1999] and also a model-elimination theorem prover in an implementation of ATP at Stanford University.

Explanation in general is one of the strengths of description logics as opposed to some of the other configuration approaches. It may be much more difficult to explain a line of reasoning in a typical constraint-based approach than it is to filter and prune an inference rule based theorem prover such as a description logic prover. Filtering object presentations and explanations in description logics has also been addressed in [McGuinness, 1996; Borgida and McGuinness, 1996; Baader *et al.*, 1999a]. Also, it has been argued elsewhere [McGuinness and Patel-Schneider, 1998; Brachman *et al.*, 1999] that explanation is a requirement for many kinds of applications, but is particularly important for configuration systems [McGuinness and Wright, 1998a].

Recent work has been done in constraint-based approaches that starts to address explanation in constraint-based configurators. While progress is being made, the more interesting constraint-based explanation systems [Freuder *et al.*, 2001] utilize extensive domain specific information and are not generic solutions to the problem of understanding explanations.

12.2.5 Evolving specifications

In many common configuration scenarios, a user begins with an incomplete set of specifications for an end product. Configuration applications built to support users should take input of the known specifications (in an order that is convenient for the user and not just an order convenient for the program), and then solicit remaining required input.

A configurator system should allow mixed initiative input - where the user may input the specifications the user is aware of at a particular time and the system should request input that it needs to meet a task. Description logics can allow users to input descriptions of end products or individual components at any time. For example, in the home theatre system, a user could specify information about the entire system—such as a requirement for the entire system to be high quality—and also could specify information about any of the particular components that she knew about at a particular time. The user might, for example, prefer to buy a particular model television or might want to set a diagonal size and a number of other constraints on the television however may not know anything at the moment about the restrictions on the DVD player.

A user interface, such as the one depicted in the stereo example, allowed a user to choose components from drop down menus. The drop down menus were generated on the fly in order to take into account all of the information that the system currently had about a component. This was used as a query to the database of all components that met that specification. Thus, the user was kept from choosing many components that would be incompatible with the system that was configured to date.

The user could also browse the current configuration and delete any requirements that were stated. (The user was not allowed to delete requirements that were inferred, however the user was allowed to ask how a particular requirement was deduced, thereby discovering the source of that requirement.) Once a requirement was deleted, then new drop down menus were generated to include components that met the current set of specifications instead of the previous set.

This architecture provides a great deal of flexibility for incrementally evolving (sometimes non-monotonically evolving) specifications. It worked well to provide users with menus of choices that were recalculated on an as needed basis with updated component lists that meet the current specifications that were stated or implied about any component.

For example, if a user stated that she wanted a high quality stereo system and then decided to choose an amplifier for the system, the configurator would only present options for amplifiers that had been determined to be high quality. Description logics are not the only modeling scheme that support evolving specifications, but this section attempts to point out that they can be used rather easily to support evolving configuration specifications.

12.2.6 Extensible schemas

Many configuration applications find that information about components is continually updated. It is not always the case that simple data about components is updated but sometimes properties of the components change or new properties are discovered after an application has been encoded. Thus, it becomes important to work with a schema or a description of a component that can be updated. For example, in our home theatre application, when we began development, DVD players were not in the consumer market. It later became common for home theatre systems to include DVD players, thus our schema needed to be extended with the new class - DVDPlayer - as well as with roles that were appropriate for DVD players.

This need for updatable and configurable schemas is sometimes a requirement for design. For example, in AT&T evaluation of software, one criteria is extensible schemas. Our experience in the deployed PROSE and QUESTAR configurator family was that products were extended often in practice.

12.2.7 Reasoning for incomplete information

Many configuration specifications are almost by necessity incomplete when input initially. In large systems, it may be common for one person who may be an expert in one area to input specifications for that area while another person who is an expert in another area may update the specification later. For example, in a two person household, one person may be much more literate in audio quality and thus that person may input the requirements for speakers and another person may have more interest and knowledge in video displays, thus that person may input specifications for the television (along with its input and output requirements). It may be important to allow specification to be done across multiple sessions as well.

One would not want a configurator that could not make deductions until all of the input requirements have been presented. For example, in the stereo system, one would want a configurator that could infer the implications of the speaker restrictions on say minimum power requirements for the amplifier, even though the television specifications have not been input yet.

Description logics have been demonstrated to be useful at determining logical consequences of information even when it is incomplete. They can also be used to determine information that is still required. For example, they can determine that two speakers need to be input as parts in the parts list before the configuration can be considered complete. Thus, it is not enough to say that two high quality main speakers are required but the parts list actually needs to have the actual speakers chosen before the job is considered complete.

In the home theatre application, there was a one-pane display dedicated to showing which final component choices still remained before a configuration could be considered completed. The display could be used to view the current parts already implied and/or chosen along with the other components yet to be chosen. The other components could be clicked on to obtain the current description of the component so that a user could view what had been derived to date about that component. The application allowed a user to save a partial specification of a configuration for further requirements to be input at another point. The application also allowed a user to "complete" the configuration at any point which would force the system to make consistent decisions for remaining underconstrained components. The user could also inspect individual component choices and click on them and see a pull down menu list of alternative choices that the system could have made. The user could also click on the component and view a description of the constraints that the application had determined must hold of that component. The description of the component was what was used to query the knowledge base about components that would fit the characteristics. The description could also be passed along to another user (or another application) so that it could see what constraints had been deduced so far and then have that other user (or application) either add new constraints or make the ultimate product choice, thereby facilitating collaborative configuration.

12.2.8 Inconsistency detection

Configuration applications should minimize the chances for users to generate inconsistent specifications. The stereo configurator, for example, uses the information that can be deduced about any particular component in order to form a query to the database about possible components. This greatly limits the chances that a user may choose a component in their system that will cause an inconsistent specification to result. The deployed application did not take a greater step however before choosing to put a component on a pull down list. It did not make the hypothetical choice of the component for the user and then check to see if the remaining components that were still unspecified could be completed with a component in the database. (Of course, this would be an exponential search with the remaining components yet to be specified.) Thus, the deployed example, could still allow a user to generate an inconsistent specification—the application just made it more difficult for this to happen. The back end reasoning system was required to determine when an incremental specification became inconsistent.

Sometimes users of other deployed configurators generate a large set of constraints and want to input them into other (connected) configuration applications. Thus one additional requirement on a user friendly configurator (that is expected to interact with other configuration applications) is for the reasoner to take input constraints and determine if they are inconsistent.

Reasoners may choose different methods of handling inconsistencies. A requirement for a configuration system is that the underlying reasoner must be able to identify the inconsistency and notify the user. A helpful reasoner will also support the user by allowing her to ask how the inconsistency was deduced. The reasoner could also give the user the option to "roll-back" the specification to the last consistent state. For example, the Classic knowledge representation system required its information to be consistent, thus once an inconsistency was detected, it disallowed the last statement that generated the inconsistency (maintaining a separate error state for debugging support) and then rolled-back to the last consistent state. This was common for early description logic-based systems. Today however, description logics do not necessarily require consistent axioms to function. They may allow a set of inconsistent axioms to be input and then configurators can be built that utilize the description logic to identify if a description is satisfiable. This model of allowing inconsistent input with a user-identified check point may be a model that supports collaboration and web-oriented development most naturally.

12.2.9 Modularity

In large systems, it is important to allow multiple people to work on specifications in what appears to be a simultaneous environment. In Prose for example, care was taken to design a set of classes and roles that a number of developers could use. Multiple users were then allowed to work on specifications of different portions of the configuration information simultaneously with previously defined upper level classes and roles for their use in specifying more specific classes. When the users were finished with their particular component descriptions, loads were done to see if the the two portions interacted. This model of individual users being in charge of specific portions of the ontology while possibly one chief ontologist is in charge of the upper level ontology is not uncommon. Cycorp, for example, publishes its upper level ontology which is maintained by a core Cycorp group while many other people develop more specialized mid-level ontologies. VerticalNet also has a number of ontologies with many different authors of specific ontologies that use an upper level ontology that was maintained by a core ontology team. Description logics can be used to support such modeling with Prose being an example of one such development.

Another notion of modularity support can be considered with environmental support features. Some systems such as OntoBuilder [Das *et al.*, 2001] at VerticalNet have been built to support multiple users working on the same portion of an ontology in a more integrated manner. VerticalNet's system allows users to be notified if someone is modifying a portion of the ontology that they are using. While OntoBuilder does not have a description logic back end, its input language is quite similar to OIL [Fensel *et al.*, 2001] and thus it is not a hard task to imagine that an OntoBuilder-like system could be integrated with today's description logic systems.

12.2.10 Maintainability

Once systems are used for a long time period or are used enough so that they require support from someone other than their original author, maintainability becomes an issue. We have used examples from the stereo configurator for all of the other sections but in this section, we will draw from our experience with the Prose/Questar family of configurators. The stereo configurator has been up on the web for some years yet it has not had many maintenance requirements because it is a demonstration system that is not updated when new stereo information becomes available. However, deployed configurators typically have help desk support and require data (and sometimes schema) updates.

There are at least three components of maintenance that require some thought when planning a configurator:

- product data updates
- product specification updates
- help desk support

The first is the simplest. Typically product data requires updates over time. Simple things like prices and availability need updating and sometimes small updates are made with revisions. Typically, this kind of information is not hard to update - someone who does not know much about the encoding can typically find a way to do things such as updating price fields in many applications - whether they are description logic-based or not. Description logics may help support this requirement more than some since they are aimed at working with incomplete information (e.g., Section 12.2.7), thus updates from incomplete to more complete information are natural for DL-based systems to handle. Similarly, an object-oriented modeling scheme may make updates simpler, but this area alone would not be enough to drive a potential user to a description logic-based approach.

The second issue of updates to product specification might be viewed by a database designer as a schema update. This kind of information is typically more challenging to update in applications since it requires product specification descriptions and not just simple data changes. It could be simple requiring say a change to the range of a field, for example, possibly an age range may move from 18–65 to 18–70. Similarly, a business that used to accept only US currency may now accept other currencies, such as Euros, thereby requiring price fields to require value restriction updates. More complicated product specification updates may be done when new components become available (thus requiring someone to model the new components and their features). These types of specification updates are facilitated in description logics by the kinds of features that we noted in Sections 12.2.6, 12.2.5, 12.2.1, and 12.2.9.

The third issue of help desk support has been noted as a strength of description logic-based systems. One of the goals with the PROSE configurator systems was to allow the help desk personnel to appear to perform at a level above the amount of training they had on individual products. The enabling infrastructure toolset was to provide information to the help desk staff at the time they needed it in real time (instead of requiring them to have been previously trained on products so that they could answer questions from knowledge that they had learned instead of from knowledge that they could look up on demand).

The tools were to allow them to explain any of the deductions that the system made when customers called in asking why something was (or was not) in their configuration and also allowed them to answer questions about why configurations were (or were not) valid. This was most facilitated by the functionality described in Section 12.2.4 but also by others such as Section 12.2.8. Similarly, they could answer hypothetical questions aimed at answering questions such as "what would happen if I chose component X instead of component Y in my configuration". The goal was to meet individual customer needs without requiring engineering support to answer such questions. Our claim is that it is a combination of the strengths of description logics as discussed in the previous sections that help support maintainability of the applications and in fact, help support maintainability by people who have not taken classes in description logics or knowledge representation.

12.3 The PROSE and QUESTAR family of configurators

The longest-lived and most prolific family of description logic-based configurators has been the PROSE and QUESTAR product line [Wright *et al.*, 1993; McGuinness and Wright, 1998b]. AT&T began development on configuration problems in 1988 in response to business requests for help in the streamlining of the Engineer, Furnish, and Install process. The goal in the process is to solicit a specification request from the customer through the sales process, and then engineer a solution that can be "furnished" and of course manufactured and delivered to the customer in a timely and cost effective manner. The initial goals of the project were to decrease the time from specification to installation and to minimize the impact of contradictions in the specifications and mistakes in the engineering. The initial configurator was built for a fiber optic transmission system (the FT Series G) although the initial deployment was for a digital cross-connect system (the DACS IV-2000).

The initial configurator was successful enough that a family of configurators was built around it. The history of development proceeded moving from more research involvement to more development involvement. AT&T's research division collaborated with developers in order to build the initial system. Researchers helped generate and critique the initial conceptual models and programming effort. Developers generated the initial system but with the help of interactive assistance from research. As the product evolved, project needs emerged for developer independence and an environment was produced that allowed domain knowledgeable people to input configuration rules in a language that was comfortable to them. Developers had the lead responsibility in the initial deployment with the assistance of research but in the second through seventeenth system, developers had the lead and required little assistance from research for either generation or maintenance of individual configurators. As the development environment evolved, the developers saw much less of the description logic back end—essentially the description logic back end verified input and deduced conclusions and was otherwise hidden behind the interface of the system.

There are a few points worth noting about this family of applications. First, the configurator family has shown longevity with some configurators deployed a decade after work began. Second, the majority of the generation and maintenance of the configurators was done by people who knew very little about description logics (thus showing empirical evidence that applications do not require PhDs in description logics to build and maintain them). An evolution interface was developed by domain literate developers aimed at users who knew the products but did not know description logics or sometimes computer science at all. This interface allowed users to both maintain configurators and also to generate new configurators in the same product family. Third, there is a consensus that the description logic-based approach both facilitates conceptual modeling (e.g., [McGuinness and Wright, 1998b]), and also makes maintenance much easier. Ford Motor company has also stated similar findings with its long-lived description logic-based configurator applications.

12.4 Summary

We have introduced the problem of configuration, describing briefly the nature of the problem and why many communities consider it important. We have described properties inherent in the problem that make it an area for which one might consider description logic-based approaches. We have provided examples of all of properties in the setting of a stereo configurator, mentioning how a description logic-based approach was used to solve the problem. We made parallel connections to the much larger configurators used for telecommunications equipment that also included the same issues and had description logic-based solutions.

We have also introduced the largest family of description logic-based configurators—the PROSE/QUESTAR family of systems (noting also that at least one other commercial configurator at Ford Motor company also has a similar life-span and a similar description logic-based approach). We observe that the PROSE/QUESTAR configurator family has been in continuous use for over a decade and has configured billions of dollars of equipment. We finally note that the commercial configuration examples with long histories state the the description logic approach has made the problems of conceptual modeling and configurator maintenance less problematic. Additionally, we speculate that this general architecture that meets the list of configuration needs might also be used in problem areas with similar needs.

Acknowledgements

While of course all errors in the chapter are the responsibility of the author, the chapter has been enhanced due to contributions from a number of people. Much of the discussion has been in the setting of either PROSE/QUESTAR or CLASSIC work that has been done in conjunction with Jon Wright, Lori Alperin Resnick, Peter Patel-Schneider, Ron Brachman, Alex Borgida, Charles Isbell, Elia Weixelbaum, Gregg Vesonder, Harry Moore, Pat Saleh, Charlie Foster, Chris Welty, Matt Parker, and a number of other important contributors. Also, Nestor Rychtyckyj provided many valuable comments on a previous version of this paper.