Developing Troubleshooting Systems Using Ontologies

Reza Basseda*, Michael Kifer*, John P. Kane†, Steven Greenspan‡ and Paul I. Fodor*
*Computer Science Department, Stony Brook University
Stony Brook, NY, 11794, USA
Email: {rbasseda,kifer,pfodor}@cs.stonybrook.edu
‡CA Inc., Islandia, NY, 11749, USA
Email: {john.kane,steven.greenspan}@ca.com

Abstract—Development of troubleshooting software is an attractive area of research for agent based system developers. In this project, we attempt to use ontologies extracted from different textual resources to automatically construct a troubleshooting virtual expert. In our solution, we verify the information about the structure of the system extracted from the textual document, then generate a conversation with the user in order to identify the problem and recommend appropriate remedies. To illustrate the approach, we have built knowledge base for a simple use case. A special parser generates conversations that can help the user solve software configuration problems.

I. INTRODUCTION

Troubleshooting a complex systems is a logical and systematic search for the sources of problems. After finding the problem sources in a complex system, the troubleshooting system provides remedies to solve the problem, so the system can be made operational again. Troubleshooting techniques are used widely in different complex systems such as smart phone services and applications. A troubleshooting and diagnosis system runs a troubleshooting process based on its knowledge about the structure and behavior of a complex system. A troubleshooting process not only identifies malfunctions within a failed system but also requires confirmation that the solution restores the failed system to its working state. Information extracted from resources on the world wide web can be considered as a potential source of knowledge to build a diagnosis system for commonly used applications and technologies such as smart phone services. The availability of various information resources on the web emphasizes the role of consistency checking of the knowledge base components for troubleshooting systems.

There are many feasible approaches based on the behavioral knowledge about the system under diagnosis. A diagnostic system based on a simple search of symptoms and causal models is presented in [1]. In [2], Portinale uses the formalism of Petri nets to provide a diagnostic model. A model-based diagnosis model for discrete event systems with an incomplete system model has been proposed in [3]. A similar model-based reasoning system with uncertain observations has been presented in [4]. In [5], Zhang et al. present a value propagation model and an algorithm for finding a minimal diagnosis. All of these approaches rely on behavioral knowledge about the system under diagnosis. In all of those methods, a behavioral model of the complex system is required, in order to be able to find problem causes. The more concrete behavioral model representing and simulating the behavior of the original system, the better resultant diagnosis system to accurately address the potential problems. Developers need deep knowledge about the behavior and structure of a system to build such behavioral model.

Several case-based reasoning models are also provided for troubleshooting and diagnostic systems. Case-based reasoning for diagnosis of mechanical systems has been presented in [6] and [7]. Bandini et al. present a diagnosis system combining model-based and case-based reasoning systems in [7]. The architectures of diagnosis systems combining case-based and model-based reasoning approaches are also considered in [8]. As those systems mainly rely on comparison between problem cases, the structural information does not play significant role in them. On the other hand, the number of cases stored by a diagnosis system directly affects the accuracy of diagnosis in such systems.

Using web data to construct a behavioral model is more difficult than developing a case based model. Development of a behavioral model of a complex system needs the knowledge of how different components of the complex system work and interact with each other in detail. Extraction of such knowledge out of various web data resources(e.g. online forums and message lists) is difficult. In fact, we often need the design documentations of a complex system to obtain such information. Such documentation is also not usually available on the web. Although building an organized repository of cases using web data is simpler than building a behavioral model, checking the consistency between the various web data resources is still crucial. In order to have a consistent knowledge base of cases, a logical representation of the system is required. As none of the above mentioned case-based approaches are based on a formal representation of the system, they are not able to completely verify the correctness of each web data resource. This leads to a challenging integration problems for different resources and directly affects the accuracy of the developed diagnosis systems.

A structural representation of a complex system can be used to organize the case-based repository of its diagnosis system. We can use such representation to verify and check the consistency of the knowledge resources. In this paper, we show how structural representation can contribute to the design of a diagnosis and troubleshooting system. Using ontologies in a particular technological domains, we automatically build a search algorithm which defines the troubleshooting process.
We also use such ontologies to verify different information resources on the web. We will also show how extracted problem/solution pairs can be shared among different classes. Basically, we will show how we can establish a troubleshooting and diagnosis process based on this ontological representation in either a procedural programming language (e.g. Java) or declarative programming language (e.g. Prolog). In our approach, we instantiate a hypothetical instance of the complex system and define the troubleshooting steps on its components.

The rest of the paper is organized as follows. Section 2 briefly describes our methodology. Section 3 explains our logical formalism for the structural representation of complex systems. Section 4 briefly describes how we produce an interactive troubleshooting process out of our logical formalism and the last section concludes the report.

II. Methodology

In this project, we use a declarative formalism to represent the structure of a complex system. This formalism is similar to description logic, as it specifies ontologies in terms of classes and objects of complex system. We use an ontology which represents the relationship among those classes and objects. This ontology plays three key roles in our method. The first use of this ontology is for evaluation of information provided by different web data resources. Since each problem-solution pair should be associated with a component in the structural representation of the complex system, problems that cannot be categorized by the ontology are not stored in our case based repository. The second use of the ontology is the expansion of the case-based repository based on the ontology. We can use relationships defined between classes in the ontology to expand the number of problem-solution pairs. The third use of our ontology is in the development of the troubleshooting process. We use use this ontology to build a search process that checks different components of the complex system to find the problem causing the failure. This search process can be either generated in a procedural programming language or provided as a general logic program with knowledge base system.

Based on the above mentioned applications of the ontology, we propose a simple process to develop troubleshooting systems out of web data. As shown in Figure 1, this process is a pipeline showing the life cycle of the troubleshooting data. In this process, we show how original troubleshooting data in web pages and forums can result in a troubleshooting procedure.

The first step of this process is to establish an ontology of the complex system and its interaction with the environment. Basically, we use a declarative language based on description logic to represent an ontology that specifies system’s components and their relationships in the complex system. Since this formalism consists of only a set of classes and relationship between these classes, it represents a static image of the complex system and does not describe how the complex system works. As the ontology does not explain the dynamic behavior of components, we can find required information to build such formalism in the web data. We use Coherent Description Framework (CDF) in XSB[9] as a logical engine for our ontology management system to keep and manage our ontology. CDF provides a logical basis and a reasoning framework that enables us to check the consistency of our ontology.

CDF provides two classes of instances: Type-0 and Type-1. Type-0 instances are suitable for storing large amounts of information because consistency and implication in Type-0 instances is computable in polynomial time. In Type-0 instances, classes are described by existential and universal relations, qualified number restrictions, and relational hierarchies. However Type-0 instances cannot express negation and disjunction defined in description logic. Type-0 also allows direct product construction for objects and classes. We also use XSB to query the ontology. There are also modules for efficient consistency checking for Type-0 instances.

Type-1 instances are equivalent to expressive description logic that supports negation and disjunction. Since our ontologies usually do not have formulas containing negation and disjunction, we do not use Type-1 instances in our application.

In addition to reasoning and representation features, CDF also provides two features that are very useful in our application. CDF has various efficient mechanisms for ensuring consistency of objects and classes in a Type-0 instance. These levels of consistency can be checked during various operations on the CDF instance. This feature is mainly used when we want to add new facts to our ontology. This process is the main part of our resource evaluation mechanism. CDF’s component system is the second feature of CDF we used in our application. The component structure of CDF let one efficiently reuse ontologies. An ontology component can be used in various troubleshooting applications.

Figure II shows a piece of our ontology for the domain of email and text message applications on smart phones. It simply states that class of smartPhone is a subclass of cellPhone and each iPhone is a smartPhone. The predicate maxAttr_ext/4 shows the cardinality restriction of hasOs relation on these classes while hasAttr_ext/3 and allAttr_ext/3 are the classes associated with hasOs relationship. There are several built-in classes and relationships provided by CDF which helps us at this stage. For example, basic data types such as numbers and integers are provided by CDF. The predicates cid/2 and rid/2 define class and relation identifiers whose second argument shows the component identifier.

We define a logical formalism that customizes the description logic to our problem. We extend the original description logic with concepts for problems and solutions. The description of our formalism is not in the scope of this paper but the interested readers can find it in our other papers.
The second stage of this pipeline is building the case repository. As mentioned before, each case describes both the problem and its solution for a particular component. According to this definition, we need to associate each case with a single component in the ontology and specify the condition for when the problem arises in that component. Each problem/solutions pair is associated with a class which is inherited by all of its subclasses. This feature enables us to expand our case repository using our ontology. Let us have an example. Assume that an information resource \( R_1 \) associates class iPhone with a problem memoryDamaged and a set of solutions. With two subclasses iPhone3G and iPhone4G of the class iPhone, the problem memoryDamaged will be inherited to both of them. This implies that our knowledge base will associate both of those classes with the memoryDamaged problem and its corresponding solutions, even though none of our information resources mention such problem for those classes explicitly. In this way, the inheritance relationship enables us to expand our knowledge about potential problems.

Note that the concept of problem and solutions are not considered as attributes of classes and objects since they are directly used to define the troubleshooting process. The separation of problem and solution concepts from attributes gives us the flexibility of implementation method for the troubleshooting process as we can directly use the set of problem/solution cases in a troubleshooting system with a declarative programming language such as Prolog. We can simply show that if our diagnosis system and generates pseudo code of the troubleshooting search process in either procedural programming language or logic programming frameworks. In our project, we have implemented a parser that gets the knowledge base of the diagnosis system and generates pseudo code of the troubleshooting search process in Java. However, it would be easy to implement that search process in declarative programming languages such as Prolog. We can simply show that if our diagnosis system knowledge base has a definitorial TBox\(^1\), we can build a loop-free search procedure. We also suppose that the ABox generated in previous steps of our pipeline does not contain any individual instances. These assumptions enables us to define the process of checking components.

Choosing to build such search procedure out of our knowledge base, we first build a single hypothetical instance of the complex system based on our knowledge about the structure of the complex system. This hypothetical instance acts as the failed system for us. We can automatically build such an instance out of the main classes. However, in our current implementation, we do not build it automatically. Finally, our parser takes these hypothetical instances and the knowledge base and generates the corresponding search pseudo code in Java. The simple program shown in Figure II is a template for a troubleshooting search procedure. Basically, it starts with the class representing the whole complex system and checks all of the conditionals defined for its own problems. Then it checks the sub-components, and this process is repeated recursively.

This three step pipeline consumes the troubleshooting data as an input and outputs a troubleshooting process. According to the method above explained, a formalism for the required complex system ontology can be defined.

\(^1\) We assume that the reader is familiar with the terminology of Description Logic.
III. EXPERIMENTS AND RESULTS

We have implemented a prototype of diagnosis system for email, text, and phone call services for iPhones using the described method. In the first step, we used a set of reliable online resources such as web forums and manuals to build an ontology of components of the original system. We then use that ontology to bind the problem/solution cases gathered by the step of information gathering. Finally, we use the knowledge base to develop the search procedure for the troubleshooting process. We have used XSB [9] and XSB’s Coherent Description Framework (CDF)[10] package to implement the code generator module of our system. This particular knowledge base is based on the artifacts delivered for the Virtual Expert for iPhone Services Project developed by CA Technologies Inc. in their innovation center in New York State Center of Excellence for Wireless and Information Technology (CEWIT). Our knowledge base has ~500 extensional facts in the main class components and 900 entries for the problem case repository. We generate Java based pseudo code in the output which will be used as a basis for a diagnosis system.

In order to highlight the role of ontology in the expansion of our problem/solutions case repository, we conducted a simple experiment. In the experiment, we compared the number of available documents for a simple case of problem on two different levels of classes in our case study. Assume that we search our resources to find potential problems for the wifiConnection attribute of class iPhone. Our experiment shows that among our first 10 search results, we can share 10% of the reflected problem/solutions from the wifiConnection attribute of class iPhone to the corresponding attributes of its subclasses, iPhone3 and iPhone4. Our results also show that 40% of those results directly point to the corresponding attributes of those subclasses. Figure 4 shows how the problems reflected on the resulted documents corresponds to the classes.

IV. CONCLUSION

In this paper, we described a development process for diagnosis and troubleshooting systems. We utilized an ontology containing structural information about complex systems to build a diagnosis system. This ontology just provides a picture of a complex system structure, which let us extract a more organized and reliable case repository out of unstructured data sources. This organized data extraction plays a key role when we are working with web data resources. We have developed a system that gets the knowledge base and builds the troubleshooting procedure in an imperative programming language. Using a practical example about smart phone services, we can use ontologies to refine and organize case base repositories for diagnosis and troubleshooting systems. We can add probability factors not only to predicates binding problems to concepts, but also to those relating solutions and problems.

ACKNOWLEDGMENT

The authors would like to thank Dr. Rong Zhao from Center for Dynamic Data Analysis, Dr. Senlin Liang from Web-Scale Knowledge Representation Lab. of Stony Brook University, and Russell Arzt from CA Technologies for their helpful advises and supports.

REFERENCES