A Logical Foundation For Troubleshooting Agents

Reza Basseda\textsuperscript{1}, Paul Fodor\textsuperscript{1}, and Steven Greenspan\textsuperscript{2}\textsuperscript{**}

\textsuperscript{1} Stony Brook University, Stony Brook, NY, 11794, USA
\textsuperscript{2} CA Inc., Islandia, NY, 11749, USA

Abstract. Intelligent software agents interacting with users arise in different applications. One of the applications of such agents, which are called virtual experts, is troubleshooting systems. In this project, we are trying to use different available textual resources to automatically construct a troubleshooting virtual expert. In our solution, we extract the information about the structure of the system from 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 a knowledge base for a simple use case. A special parser generates troubleshooting conversations that guides the user solve configuration problems.

Keywords: Description Logic, Ontology, Troubleshooting Systems

1 Introduction

Troubleshooting of complex systems is a form of problem solving, often applied to repair failed systems composed of different components and sub-systems. Troubleshooting is a logical, systematic search for the source of problems so that they can be fixed and 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 identifies the malfunction(s) within a failed system according to its knowledge about the original complex system, and it provides solutions for the potential problems. A troubleshooting and diagnosis system runs a troubleshooting process which not only identifies the malfunction(s) within a failed system, but also requires confirmation that the solution restores the failed system to a working state. An efficient and powerful knowledge representation component is usually required to have such troubleshooting and diagnostic process.

Designing such diagnostic systems dates back to 1989. In [4], Martin et al. proposes a diagnostic system based on a simple search of symptoms and causal models. Portinale [7] provides a diagnostic model which is captured within a framework based on the formalism of Petri nets; it shows how the formalization

\textsuperscript{**} The authors would like to thank Prof. Michael Kifer from Stony Brook University and John Kane from CA Technologies for their helpful advises and supports.
of the diagnostic process can be obtained in terms of reachability in Petri net model. In [12], Zhang et al. present a value propagation model and an algorithm for finding a minimal diagnosis. All of these approaches are based on the behavioral knowledge about the system under diagnosis. Basically we need to build a behavioral model of the system that we are trying to diagnose, in order to find problem causes. The ability of the diagnosis system to accurately address the potential problems depends on the ability of the behavioral model to represent and simulate the original system. Deep knowledge about the behavior and structure of a system is required to build such a behavioral model.

There are also several case-based reasoning models provided for troubleshooting and diagnostic systems. Case-based reasoning in diagnosis of mechanical systems is used in [6] and [2]. Architectures of diagnosis systems combining case-based and model-based reasoning approaches are also considered in [2] and [8]. Basically, in a case-based reasoning approach, the problem solving paradigm emphasizes the reuse of stored cases to solve new similar problems. Since these systems are based on comparing given problems with the solved cases, the structural information is not playing significant role in such diagnosis systems. On the other hand, the number of cases stored by the diagnosis system directly affects the accuracy of the diagnosis in such systems.

Using web data to construct a behavioral model is more difficult than developing a case based model. In order to develop a behavioral model of a complex system, we usually need to know how different components of the system work and interact with each other. Acquisition of such knowledge from various web data resources (e.g. online forums and message lists) is not easy. Although we can easily build an organized repository of cases using web data, checking the consistency between the various web data resources is still crucial. In order to have a consistent knowledge base of cases extracted from various web data resources, 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.

The representation of the structure of a complex system can be used to organize the case-based repository of its diagnosis system. Such representation can also help us to verify and check the consistency of the knowledge base component. Verification requires a formal and logical foundation which describes the components of the complex system and their structural relation to each other. In this paper, we provide declarative formalism for specifying such structural representation of a complex system. Adding problem-solutions pairs associated with each component to this formalism, we will be able to build a complete knowledge base used in the diagnosis and troubleshooting system. Such formalism together with the repository of cases, allows one to build a simple pseudo code for the interactive troubleshooting procedure using.

The rest of the paper is organized as follows. The next section explains our logical formalism for the structural representation of complex systems. Section
2 Logic Of Structural Knowledge Representation

As mentioned before, we are using a declarative formalism to represent the structure of a complex system in a knowledge base. This knowledge base plays two key roles in our method. The first application of this knowledge base is verification of pairs of problem-solution cases provided by different web data resources. Since each problem case should be associated with a component in the complex system, problems that cannot be categorized by the knowledge base are not stored in our repository of cases. This provides a fair tool to judge the correctness of web data sources and available information. The second application is in the development of troubleshooting process. We can use this knowledge base to build a search process, which checks different components of the complex system to find the problem causing the failure. Based on the above mentioned applications of the knowledge base, we propose a simple process to develop troubleshooting systems out of web data [3]. According to this process of diagnosis system development, we define our formalism.

*Description Logics (DL)* were developed with the goals of providing formal, declarative meanings to semantic networks and frames, and of showing that such representation structures can be equipped with efficient reasoning tools [5]. In order to represent the structure of a complex system in a diagnosis system’s knowledge base, we need to customize such a formalism to add diagnosis specific notions. For example, we need to distinguish between a *problem* and a concept representing a class. For this purpose, we can add a particular class of identifiers to separate *problems* from *concepts*. We also need to define new concept identifiers which are the product of other concepts. This also leads to a slight extension of our language. Reasoning requirements are also limited in our application. The basic inference in this domain of application is subsumption. Determining subsumption is the problem of checking whether a concept is more general than another concept. Moreover, we need to check if the knowledge base and case based repository provide an accurate picture of the complex system. We assume the reader is familiar with basic ideas of description logic and will not explain description logic in detail. Instead we will just briefly describe what we are extending in description logic according to our requirements. We call this formalism *Attribute Logic with Problem models, ALP*. ALP is an extension of the basic DL *ALC* [9][1] with identifiers for problems and solutions, and the most widely used DL reasoning services required for troubleshooting system development.

2.1 *ALP* language

In this section, we will briefly overview the syntax and semantics of *ALP*. We skip the introduction of an important extension to *ALC*, number restrictions which is used in our application domain as well. Interested readers can find more about that in [5].

3 briefly describes how we produce an interactive troubleshooting process out of our logical formalism and the last section concludes the paper.
**ALP syntax** Similar to $\mathcal{ALC}$, the main element of the $\mathcal{ALP}$ syntax is the definition of concepts. We also need elements representing problems, solutions, and guarding statements\(^3\) which shows in which condition the problem will arise.

**Definition 1 (ALP problems and guarding statements).** Let $N_C$ be a set of concept names. The set of $\mathcal{ALP}$-problems $P$ is a set of identifiers and the set of $\mathcal{ALP}$-guarding statements $G$ is a set of conditional statements such that:

$$\forall C \in N_C, \delta(C) \subseteq P \times G$$  \hfill (1)

Intuitively speaking, $\delta(C)$ relates the components of the complex system to their corresponding problems. In Definition 1, we present a guarding statement as a general identifier used as a simple conditional statement in our code generator. If we use declarative programming language to implement the troubleshooting search process, this guarding statement will appear as a logical term. In this paper, we simply take this assumptions that those guarding statements are identifiers which semantically represent the condition of raising the problem $p$ in the concept(component) $C$.

**Definition 2 (ALP solutions).** Let $P$ be a set of $\mathcal{ALP}$-problems. The set of $\mathcal{ALP}$-solutions $S$ is a set of identifiers such that:

$$\forall p \in P, \exists S_p \subset S, \sigma(p) = S_p$$  \hfill (2)

As shown in Definition 2, each problem may have several solutions. Since each problem is bound to a specific component, its corresponding solutions are bound to its component. We can simply express the effectiveness of each solution which can be used to improve the efficiency of constructed troubleshooting process out of the knowledge base.

According to our definitions of problems, solutions, and guarding statements, we can define the syntax of $\mathcal{ALP}$.

**Definition 3 (ALP concepts).** Let $N_C$ be a set of concept names and $N_R$ be a set of role names. The set of $\mathcal{ALP}$-concept descriptions is the smallest set such that

1. $\top, \bot$, and every concept name $A \in N_C$ is an $\mathcal{ALP}$-concept description. We call them atomic concept.
2. if $C$ and $D$ are $\mathcal{ALP}$-concept descriptions and $r \in N_R$, then $C \sqcup D$, $C \sqcap D$, $C \times D$, $\neg C$, $\forall r.C$, and $\exists r.C$ are $\mathcal{ALP}$-concept descriptions.

\(^3\) Another appropriate name for such statements could be conditional statements.
Note that $ALP$-concept descriptions extends $ALC$-concept descriptions with a product of concept descriptions. In our application domain, we need such extension as we may require to define problem cases on the attributes defined on the combination of different concepts. For example, in a troubleshooting system for email application of smart phones, we define two different concepts $T$ and $C$ for telecommunication company and customer. Then, we need to define an attribute of payment for some pairs of customer and telecommunication companies. Using this extension, we simply define such attribute on $C \times T$.

**Definition 4 ($ALP$ syntax).** Given $N_C$ as a set of concept names, $N_R$ as a set of role names, $P$ as a set of problem identifiers, $S$ as a set of solution identifiers, and $G$ as a set of guarding statements, we can define an $ALP$ formula according to our above definitions of concept descriptions, problems, and solutions. If $C$ and $D$ be concept descriptions, $p \in P$, $S_p \subseteq S$, and $PG \subseteq P \times G$, we can define an $ALP$ formula as having one of the following forms:

1. $C = D$
2. $\delta(C) \rightarrow PG$
3. $\sigma(p) \rightarrow S_p$
4. $C \sqsubseteq D$

A set of $ALP$ formulas are called an $ALP$ program.

A formula of the form $C \sqsubseteq D$ is called a general class inclusion formula. We also use $\rightarrow$ to highlight the difference between regular assignment operation (=) and the result of $\sigma$ and $\delta$ mappings.

Let us have a brief example to show how this formulation can help us to express our domain knowledge. Suppose $smartPhone$, $iPhone$, $samsungGalaxy$, $networkConnection$, and $wifiConnection$ are concept names in $N_C$, $hasNetworkConnection$ is role name in $N_R$, $outOfWifiRange$ is a problem identifier in $P$, $getCloserToWifiBase$ is a solution identifier in $S$, and $distance > 10\text{m}$ is a guarding statement in $G$, then Equations 3, 4, 5, 6, and 7 will be a valid $ALP$ formulas.

$$hasNetworkConnection.smartPhne = networkConnection \tag{3}$$

$$networkConnection \sqsubseteq wifiConnection \tag{4}$$

$$smartPhne = iPhone \sqcup samsungGalaxy \tag{5}$$

$$\delta(iPhone) \rightarrow \{(outOfWifiRange, distance > 10\text{m})\} \tag{6}$$

$$\sigma(outOfWifiRange) \rightarrow \{getCloserToWifiBase\} \tag{7}$$
ALCP semantics The semantics of ALP relates its formulas to four different domains. Similar to ALC, some of the formulas just express the relationship between concepts and individuals. For instance, in our above mentioned example, Equation 3 says that all of the individuals of concept iPhone3GS has a relation of hasNetworkConnection with some individuals of concept networkConnection. On the other hand, we may have formulas containing elements from problems, solutions, or guarding statement domains. For example, Equations 6 and 7 contain distance > 10m, outOfWifiRange, and getCloserToWifiBase. It is obvious that these identifiers are not included in the domain of individuals. distance > 10m belongs to the domain of guarding statements which is a set of conditional statements. In this paper, we just assume that this set of conditional statements is a set of identifiers. However, it can be expanded through prepositional logic formalism. outOfWifiRange is also a member of problem domains which should be distinguished from the domain of individuals as its members just describe some dis-functioning behavior in a concept. Obviously, the domain of solution identifiers is separated from the domain of individuals since its members are identifiers of actions. For instance, getCloserToWifiBase is an identifier for the action of getting close to the Wifi base.

**Definition 5 (ALP semantics).** An interpretation $I = <I_D, I_P>$ consists of an ALP concept interpretation called $I_D$ and an ALP problem interpretation called $I_P$. An interpretation $I_D = (\Delta^I, \Psi^I, P^I, \iota^I)$ consists of a non-empty set $\Delta^I$, called the domain of $I$, a non-empty set $\Psi^I \subseteq \Pi^I \times \Gamma^I$ in which $\Pi^I$ is called the problem base of $I$ and $\Gamma^I$ is called the guarding statement base of $I$, a function $P^I$ that maps every ALP-concept to a subset of $\Psi^I$, and a function $\iota^I$ that maps every ALP-concept to a subset of $\Delta^I \times \Delta^I$ such that, for all ALP-concepts $C, D$ and all role names $r$;

\[
\top^I = \Delta^I
\]

\[
P^I(\top^I) = \Psi^I
\]

\[
\bot^I = \emptyset^I
\]

\[
P^I(\bot^I) = \emptyset^I
\]

\[
(C \sqcup D)^I = C^I \sqcup D^I
\]

\[
P^I((C \sqcup D)^I) = P^I(C^I) \cup P^I(D^I)
\]

\[
(C \sqcap D)^I = C^I \sqcap D^I
\]

\[
P^I((C \sqcap D)^I) = P^I(C^I) \cap P^I(D^I)
\]
\[(C \times D)^I = C^I \times D^I\]  

(16)

\[P^I((C \times D)^I) \supseteq P^I(C^I) \cup P^I(D^I)\] 

(17)

\[(\exists r.C)^I = \{x \in \Delta^I | \exists y \in \Delta^I \text{s.t.} (x, y) \in r^I \land y \in C^I}\]  

(18)

\[(\forall r.C)^I = \{x \in \Delta^I | \forall y \in \Delta^I \text{s.t.} (x, y) \in r^I \rightarrow y \in C^I\}\]  

(19)

We say that \(C^I(r^I)\) is the extension of the concept \(C\) (role name \(r\)) in the interpretation \(I\). If \(x \in C^I\), then we say that \(x\) is an instance of \(C\) in \(I\).

An interpretation \(I_P = (\Pi^I, \Sigma^I, S^I)\) consists of a non-empty set \(\Pi^I\) called the problem base of \(I\), a non-empty set \(\Sigma^I\) called the solution base of \(I\), a function \(S^I\) that maps every \(\text{ALP}\)-problem to a subset of \(\Sigma^I\).

A finite set of GCIs is called a TBox. An interpretation \(I\) is a model of a GCI \(C \subseteq D\) if \(C^I \subseteq D^I\) and \(P^I(C^I) \subseteq P^I(D^I)\); \(I\) is a model of a TBox \(T\) if it is a model of every GCI in \(T\).

We use \(C \equiv D\) as an abbreviation for the symmetrical pair of GCIs \(C \subseteq D\) and \(D \subseteq C\). An axiom of the form \(A \equiv C\), where \(A\) is a concept name, is called a definition.

Note that Equations 18 and 19 do not have equivalent relations on \(P^I\) as role restrictions do not specify any control on the bound problems and guarding statements.

As \(\text{ALP}\) is an extension of \(\text{ALC}\), an \(\text{ALP}\) knowledge base is made up of two parts, a terminological part (called the TBox) and an assertional part (called the ABox), each part consisting of a set of axioms. The most general form of TBox axioms are so-called general concept inclusions. A TBox \(T\) is called definitorial if it contains only definitions, with the additional restriction that (i) \(T\) contains at most one definition for any given concept name, and (ii) \(T\) is acyclic [1]. TBox in \(\text{ALP}\) KBs are definitorial. Given a definitorial TBox \(T\), concept names occurring on the left-hand side of such a definition are called defined concepts, whereas the others are called primitive concepts. In a definitorial TBox, the extensions of the defined concepts are uniquely determined by the extensions of the primitive concepts and the role names. From a computational point of view, definitorial TBoxes are interesting since they may allow for the use of simplified reasoning techniques and reasoning with respect to such TBoxes is often of a lower complexity than reasoning with respect to a general TBox [1].

The ABox can contain two kinds of axiom, one for asserting that an individual is an instance of a given concept, and the other for asserting that a pair of individuals is an instance of a given role name. We expand the domain of function \(P^I\) which also maps individual to the problem base \(\Pi^I\).

**Definition 6 (Relations between individuals in \(\text{ALP}\)).** An assertional axiom is of the form \(x : C\) or \((x, y) : r\), where \(C\) is an \(\text{ALP}\)-concept, \(r\) is a role
name, and x and y are individual names. A finite set of assertional axioms is called an ABox. An interpretation \( I \) is a model of an assertional axiom \( x : C \) if \( x^I \in C^I \) and \( P^I(x^I) = P^I(C^I) \), and \( I \) is a model of assertional axiom \( (x, y) : r \) if \( \langle x^I, y^I \rangle \in I \); \( I \) is a model of every axiom in \( A \).

**Definition 7 (A knowledge base of ALC).** A knowledge base (KB) is a pair \( (T, A) \), where \( T \) is a TBox and \( A \) is an ABox. An interpretation \( I \) is a model of a KB \( K = (T, A) \) if \( I \) is a model of \( T \) and \( I \) is a model of \( A \).

We will write \( I \models K \) to denote that \( I \) is a model of a KB \( K \). We need to define a special predicate on concepts to show the most specific concepts in a knowledge base. This predicate will be used when we define the troubleshooting process.

**Definition 8 (A most specific concept in a knowledge base).** A concept \( C \) in an interpretation \( I \) is a most specific concept (MSC) when we have:

\[
\forall C' \in \Delta^I, C' \subseteq C \rightarrow C' \equiv C
\]

\[(20)\]

### 2.2 Inference in ALC

We define inference problems with respect to a KB consisting of a TBox, an ABox, and with a repository of problem-solutions.

**Definition 9 (Inference in ALC).** Given a KB \( K = (T, A) \), where \( T \) is a TBox and \( A \) is an ABox, \( K \) is called consistent if it has a model. A concept \( C \) is called satisfiable with respect to \( K \) if there is a model \( I \) of \( K \) with \( C^I \neq \emptyset \) and \( P^I(C^I) \neq \emptyset \). Such an interpretation is called a model of \( C \) with respect to \( K \). The concept \( D \) subsumes the concept \( C \) with respect to \( K \) (written \( K \models C \sqsubseteq D \)) if \( C \sqsubseteq D \) holds for all models \( I \) of \( K \). Two concepts \( C, D \) are equivalent with respect to \( K \) (written \( K \models C \equiv D \)) if they subsume each other with respect to \( K \). An individual \( a \) is an instance of a concept \( C \) with respect to \( K \) (written \( K \models a : C \)) if \( a^I \in C^I \) and \( P^I(a^I) = P^I(C^I) \) holds for all models \( I \) of \( K \). A pair of individuals \( (a, b) \) is an instance of a role name \( r \) with respect to \( K \) (written \( K \models (a, b) : r \)) if \( (a^I, b^I) \in r^I \) holds for all models \( I \) of \( K \).

For a DL providing all the Boolean operators, like \( ALP \), all of the above reasoning problems can be reduced to KB consistency. For example, \( (T, A) \models a \) iff \( (T, A \cup \{a : \neg C\}) \) is inconsistent.

### 3 Generating the troubleshooting process

In the last step of our diagnosis system development process, given a consistent knowledge base of concepts and potential defects, we build a search procedure which checks every component of a failed system to find the potential problem
and resolve it. In order to build the above mentioned search procedure, we assume that our diagnosis system’s knowledge base has a definitorial TBox. This assumptions helps us to simply build a loop-free search procedure. It also simplifies the structure of our knowledge base. 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 will act as the failed system for us. Then we build a search process to traverse over the components of this hypothetical instance and check for the potential problem existence. Our code generator is a simple parser which gets the knowledge base as an input and build a troubleshooting search procedure in java. The simple below pseudo code presents a template for troubleshooting search procedure. It starts with the class representing the whole complex system and check all of the guarding statements defined for its own problems. Then, it checks its sub components and this process will be repeated recursively.

A general template for troubleshooting search

```java
function troubleshooting (KB, I){
    IN = individual(I);
    while( IN.hasItem() ){
        i = IN.pop();
        check(i);
        J = children(i);
        troubleshooting(KB, J);
        IN = IN - J;
    }
}
```

Using above described method, we have implemented a diagnosis system for email, text, and phone call services of iPhone brand smart phones. We have used XSB [11] prolog to implement the code generator module of our system. We also have used XSB’s Coherent Description Framework (CDF)[10] package to develop our knowledge base. Using CDF semantic and some small prolog modules, we could develop our required knowledge base. The knowledge base is developed based on artifacts delivered for the Virtual Expert for iPhone Services Project which is provide 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 about 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 that will be used as a basic code for a diagnosis system.

4 Conclusion

In this paper, we proposed a logical foundation to build and design an elegant knowledge base for diagnosis and troubleshooting. This knowledge base provides
a picture of complex system structure which helps the diagnosis systems to extract an organized case repository out of unstructured data sources. Our formalism is a Description Logic extension including problem-solution cases. We not only extended Description Logic according to the requirements of diagnosis system, but also limited its semantics to fit in our application. We have completely developed a parser which gets the knowledge base and builds the troubleshooting procedure in an imperative programming language. Using a practical example about smart phone services, we have shown that our formalism can completely express the required knowledge about complex system structure and use it to refine and organize its case base. Our implementation also showed that this knowledge base is expressive enough to automatically build the interactive troubleshooting process.

Although constructing a behavioral model of a complex system needs to have a detail knowledge about how the complex system works, it can be really helpful to use such dynamic behavior representation beside of our structural knowledge about the complex system.

References

1. Baader, F., Sattler, U.: Description logics with aggregates and concrete domains. Inf. Syst. 28(8), 979–1004 (Dec 2003), http://dx.doi.org/10.1016/S0306-4379(03)00003-6