A Constraint-Based Approach to Diagnosing Software Problems in Computer Networks

Daniel Sabin, Mihaela Sabin, Robert D. Russell and Eugene C. Freuder

Department of Computer Science, University of New Hampshire, Durham, NH 03824 USA

Abstract. Distributed software problems can be particularly mystifying to diagnose, for both system users and system administrators. Model-based diagnosis methods that have been more commonly applied to physical systems can be brought to bear on such software systems. A prototype system has been developed for diagnosing problems in software that controls computer networks. Our approach divides this software into its natural hierarchy of layers, subdividing each layer into three separately modeled components: the interface to the layer above on the same machine, the protocol to the same layer on a remote machine, and the configuration. For each component knowledge is naturally represented in the form of constraints. User interaction modeling is accomplished through the introduction of constraints representing user assumptions, the finite-state machine specification of a protocol is translated to a standard CSP representation and configuration tasks are modeled as dynamic CSPs. Diagnosis is viewed as a partial constraint satisfaction problem (PCSP). A PCSP algorithm has been adapted for use as a diagnostic engine. This paper presents a case study illustrating the diagnosis of some problems involving the widely used FTP and DNS network software.

1 Introduction

One of the fundamental problems confronting users and managers of computer networks today is the diagnosis of problems arising within the network itself. The symptoms produced by such problems are often baffling since they are so unpredictable and so unrelated to the task for which the network is being used. Furthermore, the error messages from the system are usually so general and vague that little can be gleaned from them as to the exact cause of (and hence the fix to) the problem. Diagnosing under these circumstances is currently more art than science [14]. The situation has been summarized in a cartoon that pictures a visitor to a computing site staring at a swami sitting cross-legged in the corner, and receiving the explanation: "That's our network guru".

Techniques for model-based diagnosis have been used successfully in the diagnosis of physical systems [10]. We have applied and extended this approach to computer network software. We consider this software to be constructed in a *hierarchy of layers* fashion as described in the ISO OSI Reference Model [12].

We subdivide each layer into three separately modeled components: the interface to the layer above on the same machine (or to the user, in the case of the application layer), the protocol to the same layer on a remote machine, and the configuration. This decomposition is developed further in the paper.

Our approach considers diagnosis as a dynamic partial constraint satisfaction problem. Activity constraints are used to interface the model with the "real world" of the network, allowing the model to dynamically obtain data from the network and to use that data to change the problem as the search for a solution progresses. The partial solutions discovered by our system constitute the diagnosis.

The next section concentrates on individual components, presenting examples of problems from widely used *File Transfer Protocol* (FTP) and *Domain Name Service* (DNS) software. Sect. 3 presents the theoretical background for our approach. Sect. 4 gives some details on how problems are represented in our system, presents the actual dynamic partial constraint satisfaction algorithm and shows how one of the sample problems, previously presented in Sect. 2, is actually diagnosed by this algorithm.

This work concentrates on modeling the network *infrastructure* itself. We believe it gives us a solid foundation for understanding basic problems that occur at all levels in information networks, and that the techniques we have developed are applicable at higher levels as well.

2 FTP Case Study

The application protocol chosen for exemplification is FTP, described in RFC 959 [19]. It provides interactive file transfer and relies on Transmission Control Protocol (TCP) – a transport protocol in the TCP/IP hierarchy of protocols [4].

We decompose the problem domain into three components, each of which will be modeled separately, as shown in Fig. 1.

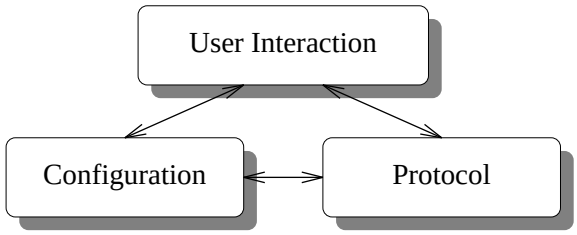

Fig. 1. Problem domain decomposition

1. The *User Interaction* component deals with commands given by the human user and the expectations that the user has about the system response to these commands.
2. The *Protocol* component is really the *network* software, and will later be further decomposed into layers corresponding to the actual implementation of the network software. For now it represents simply all *active software*.

3. The *Configuration* component represents all *passive* information about the network and the computing environment, such as machine names and addresses, routes, connectivity, etc. This information parameterizes the software in the protocol component.

The complete FTP service is structured in two distinct processes, plus the rules and formats, namely the protocol, for information exchange between them. These processes, a client (local) FTP program and a server (remote) FTP daemon, cooperate to accomplish the file transfer function. The client initiates the connection and forwards user commands to the remote server. The server, in response to the received commands, sends replies whose general format is a completion code (including any error code), followed by a textual description of the action taken. A user interface resides on the local machine through which the user requests the transfer of files to and from the remote machine. In our simple examples only two of the basic operations are used:

`put`	*local_file_name*	*remote_file_name*
`get`	*remote_file_name*	*local_file_name*

As a result, a local (remote) file is copied to (from) the remote (local) machine.

But in order to be able to communicate with the server, the client has to establish a connection with it first. Therefore, the client needs to know the server's IP address. Since people prefer names, while computers prefer numbers, programs that interface with users have to map names to numbers, and vice versa. Programs that utilize networking need to map between the names used by people to refer to host computers and the IP addresses (numbers) used to communicate over the network. The Domain Name Service (DNS) [18] consists of a method for constructing names of host computers in a hierarchical manner and a way to resolve these names in a distributed fashion. The Berkeley Internet Name Domain (BIND) is a set of procedures used to map DNS names to IP addresses and vice versa [1] [11]. It consists of a number of server daemons running at various locations in the Internet, each with the responsibility of resolving a subset of the names. On OSF/1 [5] a name is resolved as follows:

1. The file `/etc/svc.conf` is consulted to see what services are available and in what order they are to be used. Possibilities are *local* and *bind*. Each service in the list is attempted in turn until either the name is resolved or the list is exhausted, in which case the resolution fails.
2. The *local service* is provided by consulting the file `/etc/hosts` which contains a table of known names and IP addresses. Resolution consists of searching for the name in the table and it fails if the name is not there.
3. The *bind service* is provided by contacting a server daemon called `named`. If the configuration file `/etc/resolv.conf` is present, it is consulted to find an ordered list of the IP addresses of server daemons to contact. Each daemon in turn is contacted until one of them responds. Resolution fails if none of the servers responds, or if the first one that does respond is not able to resolve the name into an IP address. If the `/etc/resolv.conf` file is not present,

an attempt is made to contact a server daemon running on the local host. In this case, resolution fails if there is no such server, or if that server is not able to resolve the name into an IP address.

In the examples shown we have run the FTP program on two different machines whose operating systems are ULTRIX (a version of UNIX) and VMS. Their filesystems differ in the way files are represented and named.

According to our framework for network diagnosis we can group the FTP problems we encountered into three categories: user interaction, protocol and configuration problems. For each category we discuss some interesting examples for which we give the diagnosis solution produced by the diagnosis tool. The examples we will present are actual cases we have run and diagnosed. The format of each example explains the FTP command and its execution context, formulates the problem encountered and prints out the message produced by the diagnosis program.

To understand why the problems of the first two categories occurred, we mention briefly the VMS conventions in naming files, which are more restrictive than those for the UNIX filesystem. The name format requires three different fields for name, extension and version number. Only alphanumeric characters are allowed (plus the $ character) to specify the name field. The file extensions are predefined to indicate the type of the file. Other special characters are reserved for wildcards or delimiters in the file name syntax. Thus, names such as "@@" or "!!" are not accepted in the VMS filesystem. Choosing reserved characters to name files usually produces messages such as: "file specification syntax error", "invalid wildcard operation" or "not a plain file", which are self explanatory. However, there are cases in which the FTP server responses are rather cryptic or the transfer results are totally unexpected. We will explore these cases and explain both the FTP execution context and its effects.

2.1 Sample User Interaction Problems

The differences between the filesystems of the local and remote machine can cause surprising name mapping for the files transfered. Newly created file names on the remote machine are generated when the file specification is illegal. This is also a case of name mapping when the filesystems do not have the same naming conventions. An example is shown in Fig. 2.

There are cases in which the files get partially changed without violating naming rules. Even if the sequence of user commands shows no error, as presented in Fig. 3, examining the text files or running the executable files shows strange results. This may happen when the type of the transfer (ASCII or binary) does not match the type of the file transfered. For the ASCII type, conversion to a standard text file representation on the network is necessary to allow communication between different filesystems. The UNIX system considers the newline character as text line delimiter, while VMS uses a line length count. Thus, for the ASCII type of transfer, the sending and receiving sites perform the necessary transformations between the standard representation and their internal representation

FTP Command:	`put @@ @@`
FTP Error Message:	`---`
FTP Context:	- Local FTP running on UNIX, remote FTP running on VMS
	- Local file "@@" exists, but is not a valid file name for VMS
Problem:	The transfer takes place, but the remote name is changed to "AA.;1"
Diagnosis:	`*** Remote file name has the form "$?$?.;?"`

Fig. 2. Invalid remote file name specification

of files. It is the user's responsibility to correlate the data representation used and the transformation function performed during the transfer.

FTP Command:	`put alpha alpha`
FTP Error Message:	`---`
FTP Context:	- Local FTP running on UNIX, remote FTP running on VMS
	- Local file "alpha" exists on UNIX, and is an ASCII file
	- The transfer type is "binary"
Problem:	The content of the transfered file is not modified, since the conversion to the standard representation does not apply when "binary" transfer is used. As a result, <EOL> is not recognized on VMS and lines are incorrectly displayed
Diagnosis:	`*** Change type to ASCII and redo the transfer`

Fig. 3. Incorrect transfer type specification

2.2 Sample Protocol Problem

Sometimes illegal parameters in user transfer commands are discarded by the FTP client and the FTP server receives incompletely specified requests. The received error message has no useful meaning for the user, as shown in Fig. 4.

It is interesting to notice how the protocol knowledge helps isolate a problem triggered at the user level. In this sense, a further exploration of the underlying protocols could extend the problem domain we address.

2.3 Sample DNS Configuration Problem

We assume that all the basic configuration tasks, such as installing TCP/IP in the kernel and configuring the network interfaces and routing, have been

FTP Command: `get !! !!`

FTP Error Message: `RETR: command not understood`

FTP Context: - Local FTP running on VMS, remote FTP running on UNIX
- File "!!" exists on the remote machine, but is not a valid VMS file name

Problem: The lower level command RETR used for implementing the user GET command requires a nonempty argument. Due to an implementation error, the client sends to the server a RETR request with an empty argument. The server returns an error message which is not understood by the user.

Diagnosis:
```
*** FTP client implementation error --
    RETR request with no argument
*** Invalid VMS file name "!!"
```

Fig. 4. Protocol implementation error

performed correctly and we look only at various configurations required by the name service.

All the problems in this category follow the same simple scenario: the user tries to connect to a remote host whose name is the parameter of the `ftp` command and the connection is refused. In all cases the remote system gives the same error message: `unknown host`. However, the underlying configurations can be vastly different. For each of these erroneous configurations our diagnostician program figures out what the problem is and provides more useful messages, as shown in the example in Fig. 5.

One type of configuration problem is caused by incompletely specifying the *host table* when only local resolution is used. If the `<remote-host>` name given by the `ftp` command is missing from the `etc/hosts` file, then this host is unknown. This diagnosis is shown in Fig. 5.

Configuration: - Local resolution only indicated in `/etc/svc.conf`
- <*remote-host*> name is not in `/etc/hosts`

Diagnosis:
```
*** Local resolution failed.
    No <remote-host> in /etc/hosts.
```

Fig. 5. Local resolution with incompletely specified host table

3 Background

Model-based diagnosis techniques compare observations of the behavior of a sys-

tem being diagnosed to predictions based upon a model of the system in order to diagnose faults [10]. The fundamental presumption behind model-based diagnosis is that, assuming the model is correct, all the inconsistencies between observation and prediction arise from faults in the system. Given a model description and a set of observations, the diagnosis task is to find a set of faults that will explain the observations. Minimal diagnoses postulate sets of faulty components that are minimal in the sense that no proper subset provides an explanation.

3.1 Model-based Diagnosis as Partial Constraint Satisfaction

Constraint satisfaction is a powerful and extensively used artificial intelligence paradigm [7]. A *constraint satisfaction problem* (CSP) involves a set of problem variables, a set of values for each variable and set of constraints specifying which combinations of values are consistent. A solution to a CSP specifies a value for each variable such that all the constraints are satisfied.

If we assign costs to the values we can look for a solution with optimal cost. Model-based diagnosis can be viewed as a constraint optimization problem by associating system components with constraints that reflect their behavior, component inputs and outputs with problem variables, and introducing assumption variables associated with the system components, where a value of 0 for an assumption variable reflects normal behavior and a value of 1 abnormal behavior [6]. Observations force assignment of some of the problem variables. The task of finding a minimal diagnosis corresponds to finding an optimal solution of such a CSP.

We use a refinement of this approach based on the notion of a *partial constraint satisfaction problem* (PCSP) [8]. PCSPs were introduced for applications that settle for partial solutions that leave some of the constraints unsatisfied, e.g. because the problems are overconstrained or because complete solutions require too much time to compute.

We have found that PCSPs provide an elegant approach to viewing diagnosis in CSP terms. Regarding components as constraints, and faulty components as failed constraints, minimal diagnoses naturally correspond to PCSP solutions that leave minimal sets of constraints unsatisfied. These sets are minimal in that there is no solution which leaves only a proper subset unsatisfied. Bakker et al. [2] have taken the opposite approach, applying model-based diagnosis methods to partial constraint satisfaction.

Combinations of branch and bound and CSP techniques have been used in algorithms that search for a solution that leaves a minimal number of constraints unsatisfied [8]. We have adapted one of these algorithms to search for solutions with minimal sets of unsatisfied constraints. One of the advantages of viewing diagnosis as a PCSP is that it permits us to bring our experience with PCSP algorithms to bear on diagnosis.

3.2 Modeling Configuration as Dynamic Constraint Satisfaction

For synthesis tasks such as configuration and model composition, the constraint problem is of a more dynamic nature [15] [16] [17]. Any of the elements of the CSP might change during the search process. Mittal and Falkenhainer introduced the notion of a *dynamic constraint satisfaction problem* (DCSP) by adding a new type of constraint, called an *activity constraint*, on the variables considered in each solution. Activity constraints, expressed in terms of consistent assignment of values to some already instantiated set of variables, specify which variables and constraints should be added to or removed from the current CSP. The problem thus changes as search progresses.

The main advantage of this extension to the standard CSP is that inferences can now be made about variable activity, based on the conditions under which variables become active, avoiding irrelevant work during search.

The definitions of a dynamic constraint satisfaction problem and activity constraints, as stated in [15], are the following:

Given

- A set of variables V representing all variables that may potentially become active and appear in a solution.
- A non-empty *initial set of active variables* $V_I = \{v_1, \ldots, v_k\}$, which is a subset of V.
- A set of discrete, finite domains $D_1, \ldots, D_k$, with each domain D_i representing the set of possible values for variable $v_i \in V$.
- A set of *compatibility constraints* C^C on subsets of V limiting the values they may take on. These correspond to the standard set of CSP constraints. In addition, if any of the variables involved in the constraint are not active, the constraint is trivially satisfied.
- A set of *activity constraints* C^A on subsets of V specifying constraints between the activity and possible values of problem variables. There are four types of activity constraints, which can be divided into two groups:
  1. *require variable* and *require not*, which establish the activity (inactivity) of a variable based on an assignment of values to a set of already active variables. A require-variable constraint is logically equivalent to: $active(V_1) \wedge \ldots \wedge active(V_j) \wedge P(v_1, \ldots, v_j) \rightarrow active(V_k)$, where P is a predicate, v_i is the current value assigned to variable $V_i, \forall i, 1 \leq i \leq j$ and $V_k \notin \{V_1, \ldots, V_j\}$.
  2. *always require variable* and *always require not*, which establish the activity (inactivity) of a variable based on the activity of other variables, independent of their current value. An always require variable constraint is logically equivalent to: $active(V_1) \wedge \ldots \wedge active(V_j) \rightarrow active(V_k)$, where $V_k \notin \{V_1, \ldots, V_j\}$.

Find

- All solutions, where a solution is an assignment A which meets two criteria:

1. The variables and assignments in A satisfy $C^C \cup C^A$.
2. No subset of A is a solution.

4 Representation and Reasoning

We model each of the components of Fig. 1 – user interaction, protocol, configuration – as a separate PCSP knowledge base. Protocol diagnosis has been studied previously as a constraint satisfaction problem [3] [9] [20]. We apply a similar approach here to the FTP protocol. We demonstrate here that user interaction diagnosis can also be modeled as a constraint satisfaction problem, in particular by introducing constraints that reflect user assumptions. Finally, extending the representation used by [17], we are able to treat diagnosis of configuration tasks as a PCSP as well.

These three components are naturally modeled separately. They utilize different mechanisms to instantiate the general CSP paradigm, e.g. an intermediate finite state machine model for protocols. Applying the diagnostic engine successively to the three separate domains, until a diagnosis is found, may reduce the combinatorial complexity the engine faces. On the other hand, it is already clear that there are interesting interactions between these components, which may ultimately require a more sophisticated control architecture.

There was also a knowledge engineering, knowledge acquisition effort in developing the user interaction model. Considerable time was spent exploring different types of interaction that can occur, and discovering different types of problems that can arise.

4.1 User Interaction

The FTP commands specify the parameters for the data connection (data port, transfer mode, representation type, structure, etc) and the nature of the file system operation (store, retrieve, append, delete, etc).

Each time the user gives a command, the current state of the FTP client can be represented as a PCSP problem. The set of variables includes the transfer parameters: MODE, STRUCTURE, TYPE, the local and remote operating systems: CLIENT, SERVER, the file system operation, COMMAND, and the file pathname, PATH. In addition, there are some other variables which have no direct correspondent among the entities that characterize the state of the client. They represent instead, either the user's perception of the result of the operation, or, to some extent, the state of the user's mind at that moment. The variable OUTCOME represents the outcome of an FTP operation. Since the value of this variable cannot be determined at the time of the transfer, the user is responsible for supplying a value ("success", or, if something went wrong, his perception of "wrong", e.g. "ascii file incorrectly transferred"). Clearly, the motivation for introducing this variable in the PCSP is that it allows us to embody faulty behaviors in the model.

A "fault" at this level typically means a mismatch between the status of the real world and the user's mental representation of it. For example, data representations are handled in FTP by a user specifying a representation type, described in our model by the variable TYPE. When the user is specifying a value for the TYPE variable, he is in fact just making an assumption about the actual type of the file, represented by the value of the variable ACTUAL-TYPE. User's assumptions are modeled in a natural way with constraints. In this particular case, there is an equality constraint between variables TYPE and ACTUAL-TYPE.

This difference in semantics implies that all such PCSPs will have two sets of constraints:

1. constraints that will be part of all the PCSPs, representing the functional specification model of FTP (accounting for both correct and incorrect behavior);
2. constraints that change from problem to problem:
   (a) constraints modeling user's assumptions about the real world;
   (b) some FTP commands translate to unary constraints, forcing value assignments for the corresponding variables (e.g. the FTP command `get` restricts the domain of the variable COMMAND to a single value, namely *get*).

4.2 Protocol

Protocol specifications are typically represented in the form of finite automata, often referred to as *finite-state machines* (FSMs). Since simple FSMs have limited expressive power in representing such notions as timers, logical conditions, etc., a more powerful formalism is needed, and thus extended finite automata have been used for protocol testing and specification analysis or diagnosis [13] [21].

The idea of using model-based techniques to diagnose communication protocols based on extended finite automata is not new. To our knowledge, at least three protocol diagnosis systems have been proposed [3] [9] [20]. All these approaches attempt to diagnose protocols by analyzing conflicts between observations and the protocol model. This implies that observations must somehow be associated with the model.

The representation approach we are using is similar to the one used by Riese in [20]. The FTP protocol specification as an extended finite transducer is translated into a standard CSP form.

Where Riese is using a specialized algorithm for solving the diagnosis problem, he calls it HMDP, we are using a variant of a standard PCSP algorithm to produce the set of minimal diagnoses.

We make the same assumptions as Riese does, that the external observer resides outside of the node on which the system under diagnosis is implemented, and that the observer can time-stamp messages when they are observed.

In addition to the time stamp of the message, an observation also contains the type of the message (STOR, RETR, etc.), the corresponding arguments, if any, and the direction of the message, relative to the client (SEND or RECEIVE).

Each observation has a CSP variable associated with it. Considering the order given by the time stamps, let $OBS =< o_1, \ldots, o_n >$ be a sequence of observations and let v_i be the variable associated with observation o_i. The domain of values of such a variable is simply the set of all valid state transitions described by the extended finite transducer. Thus, to solve the CSP we have to assign one state transition of the protocol machine to each variable corresponding to an observation, subject to two kinds of constraints:

1. unary constraints, which check whether the value t_i assigned to some variable v_i has the same message type, direction and number and type of arguments as the associated observation o_i;
2. binary constraints, which relate a variable v_i to its neighbors v_{i-1} and v_{i+1} by checking whether the pairs of corresponding values t_i, t_{i-1} and t_i, t_{i+1} respectively, are part of a sequence of transitions allowed by the protocol machine. Due to transitivity, if all the binary constraints are satisfied, a solution to the problem will represent a complete transition sequence explaining OBS.

When the FTP implementation is faulty, conflicts between observations and the FTP model will result in partial satisfaction of the constraints, and the diagnosis algorithm applied to this PCSP will produce the set of minimal diagnosis in terms of errors at the level of the protocol commands (e.g. incorrect/missing arguments) and/or sets of faulty state transitions.

4.3 Configuration

We use the same approach as [15], but extend the definition of DCSP to that of *dynamic partial constraint satisfaction problem* (DPCSP), by relaxing two of the requirements in the previously presented DCSP definition.

First, we do not restrict the domains of values for variables to be predefined finite sets of values. In some cases domains are still finite sets of values, known from the beginning, but this is not always true. Due to the nature of our application, the values some variables may take are known only during the search, when these variables become active.

Second, since we are trying to solve a diagnosis problem which might have no complete solution, the (partial) solution we accept may violate some of the constraints, but we are still looking for an optimal solution, according to some criterion (e.g. minimal number of violated constraints).

Studying name service configuration and modeling it using the DCSP formalism, we found out that quite a simple language is sufficient for specifying the associated dynamic constraint satisfaction problem. Since a DCSP has four basic components, a program in this language will naturally have four sections:

1. a section defining the set of variables and corresponding domains of values,
2. a section specifying the set of activity constraints,
3. a section specifying the set of compatibility constraints, and
4. a section specifying the initial set of active variables.

Variable Specification Figure 6 presents the variable definition section for a simplified model of the name service.

```
// Variables

VAR  remote-host     ASK prompt-user("Remote host name:")
VAR  ping-path       DEF "/sbin/ping"
VAR  services-file   DEF "/etc/svc.conf"
VAR  resolve-file    DEF "/etc/resolv.conf"
VAR  hosts-file      DEF "/etc/hosts"
VAR  ping-response   ASK ping($remote-host)
VAR  resolution-type ASK resolve-service($services-file, "hosts")
VAR  hosts           ASK resolve-host($hosts-file)
VAR  local-server    DEF "/etc/named.pid"
VAR  servers         ASK resolve-name-server($resolve-file)
VAR  domain          ASK resolve-domain($resolve-file), local-host())
```

Fig. 6. Variables definition section

In order to specify a CSP using the specification language, all variables have to be declared using a VAR statement. Each variable is completely specified by the value of two attributes: *name* and *domain* of possible values.

When the domains are known ahead of time, we simply need a way to directly express them as sets of values. But since the domains of values are not predefined finite sets of values for all variables, we also need a way of specifying a procedure by which a domain will be obtained when the variable becomes active during search. Accordingly, the language offers two built-in mechanisms for specifying the domain of values for a variable:

a) The user can supply a *default*, or predefined, domain as a set of values by using the DEF slot of the VAR statement.
b) In case the domain of a variable is not known at specification time, the user must supply, as the value of the ASK slot, the call to a function which, when executed, will return the set of values in the domain. Function execution will be triggered by the activation of the variable. The function may take as arguments either constants (e.g. string, number) or the current value of variables which, at the time of the call, are already active. The current value of a variable is selected by the expression *$variable*, where *variable* is the name of the variable.

Modeling the name service configuration, we have to provide several user-defined functions which inspect configuration files, invoke UNIX system calls or prompt the user in order to get the *asked* values for the current variable. The *prompt-user* function takes one parameter, the message to be displayed. All the functions that examine configuration files need at least one parameter, indicating the name of the file where the possible values might be found. Some of them

require a second parameter, usually a string constant, to localize the line in the file where the information is stored.

Constraints Specification Activity and compatibility constraints are specified in the form of boolean expressions over variables and their possible values. The language provides the standard logical and relational operators, enhanced, for increased flexibility, with set-based operators (e.g. test for set membership, set inclusion, etc.). The operands can be constants, the current value of active variables, selected using the *$variable* expression, built–in and user–written functions, taking as arguments any of the above.

As an example, Fig. 7 presents the activity and compatibility constraints in the name service configuration model. The keywords START, ARV, RV stand for initial set of active variables (START), always require variable (ARV) and require variable (RV). When specifying a compatibility constraint, the user must also supply a formatted output statement which, in case the constraint fails, will be printed as the diagnostic message.

```
// Initial Set of Active Variables

START remote-host

// Activity Constraints

ARV remote-host ⇒ (ping-path ping-response)
RV  $ping-response = "unknown" ⇒ (services-file resolution-type)
RV  $resolution-type = "local" ⇒ (hosts-file hosts)
RV  $resolution-type = "bind" ⇒ (resolve-file servers domain))
RV  $servers = nil ⇒ local-server

// Compatibility Constraints

CON $remote-host IN $hosts
    "*** Local resolution failed. No $remote-host in $hosts-file."
CON $local-server != nil
    "*** Local resolution failed. No $remote-host in $hosts-file."
CON $servers = nil OR bind-resolve($remote-host $servers $domain)
    "*** BIND resolution failed."
```

Fig. 7. Activity and compatibility constraints definition section

4.4 Algorithm Description

Figure 8 provides a basic branch and bound algorithm for solving dynamic partial constraint satisfaction problems. It is a refinement of a partial constraint satisfaction algorithm presented in [8].

```
bound ← {{con|con is a compatibility constraint }}
algorithm BRANCH&BOUND (distance, search-path, variables, values)
   if (variables = ∅) then
      if (distance = ∅) then
         return true
      for each element D ∈ bound do
         if (distance ⊆ D) then
            bound ← bound \{D}
         □
      bound ← bound ∪{ distance }
      return false
   □
   if (values = ∅) then
      return false
   crrt-variable ← first variable in variables
   crrt-value ← first value in values
   new-distance ← distance
   subsumed ← false
   for each constraint C involving crrt-variable and variables in search-path
      until (subsumed = true) do
      if (C fails) then
         new-distance ← new-distance ∪ {C}
         if (∃ D ∈ bound such that D ⊆ new-distance) then
            subsumed ← true
      □
   if (subsumed = false) then
      required-variables ← RUN-ARV(crrt-var) ∪ RUN-RV(crrt-var, crrt-val)
      new-variables ← variables \{ crrt-variable }∪ required-variables
      if (BRANCH&BOUND(new-distance,
         search-path ∪⟨ crrt-variable, crrt-value ⟩,
         new-variables, domain of first variable in new-variables))
      then
         return true
   □
   return BRANCH&BOUND(distance, search-path, variables, values \{ crrt-value })
□
```

Fig. 8. Dynamic partial constraint satisfaction algorithm

Branch and bound operates in a similar fashion to backtracking in a context where we are seeking all solutions that violate minimal, under set inclusion, sets of constraints. The algorithm basically keeps track of the best solutions found so far and abandons a line of search when it becomes clear that the current partial

solution cannot lead to a better solution. In fact, the notion of failure during search is the main difference between CSP and PCSP. A CSP search path fails as soon as a single inconsistency is encountered. A PCSP search path will fail only when enough inconsistencies accumulate to reach a cutoff bound.

The *bound* in our context is a set containing the sets of constraints left unsatisfied by the best solutions found so far. If at any time during the search the set of constraints violated by the current partial solution, which we call the *distance*, becomes a superset of any element in the *bound*, the current search path is abandoned.

Once the search path is complete, i.e. all variables have been assigned a value, if its *distance* is a subset of any element in the *bound*, then that element will be replaced by the *distance*. In other words, the partial solution we found is better than a previous solution in the sense that it violates only a subset of the constraints violated by the previous solution.

The search process stops when either a complete solution, one that satisfies all the constraints, is found, or when we exhausted all the values for all the variables. Finding a complete solution is equivalent, from the diagnostic point of view, to finding that the configuration under diagnosis is correct, i.e. it presents no "faults". In the second case, the *bound* represents exactly the set of minimal diagnoses, that is, the set of minimal sets of constraints, one for every "best" partial solution found.

For the sake of simplicity in presentation, the algorithm in Fig. 8 does not in any way use the partial solution it finds (*search-path*). In fact, each element in the *bound* is not only a set of constraints, but a pair: set of constraints and the corresponding partial solution.

When we presented the language, we said that the definition of each constraint includes an output statement, which represents the text of the diagnostic message, in case the constraint is violated. When the algorithm stops, each element in the *bound* represents a possible minimal diagnosis for the configuration being tested. Therefore, one diagnostic message will consists of all the strings included in the definitions of the constraints in one such element.

The algorithm can also produce, if requested by the user, an explanation for each diagnosis, by printing the values assigned to each variable in the corresponding search path.

4.5 Sample Trace

We show in Fig. 9 a trace of our algorithm solving the problem presented in Fig. 5. Initially, only variable *remote-host* is active. Since it has an ASK function of type PROMPT-USER, the user will be asked to provide the name of the remote host. Let's say the user typed in `xx.xx.xx`. Due to the ARV constraint, variables *ping-path* and *ping-response* become active (STEP 1). Using function PING, the value of *ping-response* is set to "unknown". One of the RV constraints is satisfied now and variables *services-file* and *resolution-type* are activated (STEP 2). Using function RESOLVE-SERVICE, the algorithm decides that the value of *resolution-type* is "local". A new RV constraint is satisfied. Accordingly, variables *hosts-file*

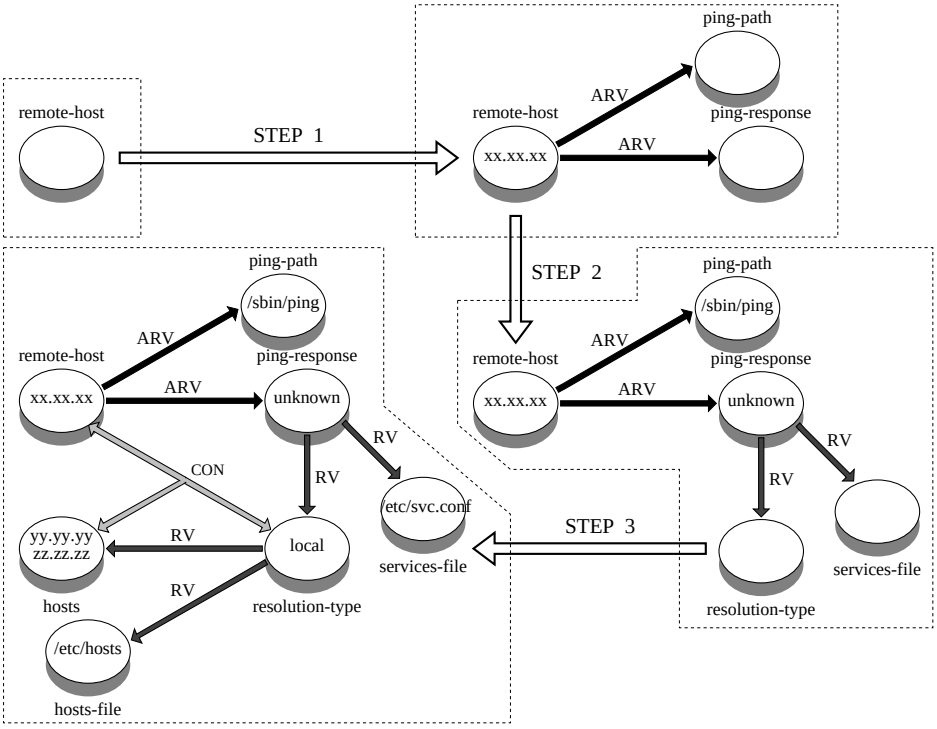

Fig. 9. Example trace for the problem presented in Fig. 2

and *hosts* become active (STEP 3). Variable *hosts* is initialized to the list of host names read from the `etc/hosts` file. Since all the variables involved in the compatibility constraint among *remote-host*, *hosts* and *resolution-type* have been instantiated, the constraint becomes active and the check fails. Because there are no other values to try, the current assignment represents the only solution of this DPCSP. So, the algorithm stops with the value of *bound* being a set with one element, the set containing only one constraint, the one that just failed. The diagnostic message is thus the string produced by the associated output statement:

```
*** Local resolution failed.
No XX.XX.XX in /etc/hosts.
```

which is the current diagnosis for this problem.

5 Conclusion

The prototype system we developed for diagnosing software problems in computer networks uses model-based diagnosis techniques. Given a model description of the software system, and a set of observations describing faulty behavior when the service is provided, the diagnosis task finds the set of errors that explain the

observations and gives precise diagnosis messages. We use a PCSP approach to view the model-based diagnosis in CSP terms, where the interacting components that define the service are the constraints. Since we solve a diagnosis problem which might have no complete solution, we need to accept partial solutions which violate some of the constraints. Thus, minimal diagnoses correspond directly to PCSP solutions that leave minimal sets of constraints unsatisfied. The dynamic nature of a configuration task is described in terms of DPCSP: at any given point in the search process configuration components are added or removed dynamically from the current problem. This enables our system to obtain current information directly from the network by applying user-written functions supplied with the model. Data thus obtained is used to guide the search by determining which components to activate. We showed the effectiveness of our prototype system on several sample problems for which more meaningful diagnosis messages have been produced.

We consider two ways in which our system could be extended, to diagnose both widely used Internet high-level services, such as NFS, NIS, etc., and lower-level protocols in the protocol hierarchy. To achieve the second goal, the mechanism used in our system is powerful enough to allow *on-line diagnosis* of lower level protocols. In our initial exploration of configuration problems we chose BIND because it is high in the protocol hierarchy, at the application level, and there are already useful tools, such as "ping" and "nslookup", that can be coupled directly into our system to provide dynamic information. However, we need to extend the problem domain to involve the entire protocol stack to detect errors that might propagate up the stack. These errors may affect the system performance or, even if an error at one level is handled properly by the protocol at a higher level, it might signal future errors. For on-line diagnosis we need to be able to run our system in a *monitoring mode*, whereby normal situations are checked in order to detect faults before they propagate. For this, we need to develop appropriate data gathering tools that filter the huge amount of data exchanged by lower-level services.

Acknowledgments

This material is based on work supported by Digital Equipment Corporation, and by the National Science Foundation under Grant No. IRI-9207633.

References

1. Albitz, P. and Liu, C., *DNS and BIND*, O'Reilly & Associates, Inc., Sebastopol, CA, 1994.
2. Bakker, R.R., Dikker, F., Tempelman, F. and Wognum, P.M., Diagnosing and solving over-determined constraint satisfaction problems, *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, **1**, 276–281, 1993.
3. Bouloutas, A.T., Modeling Fault Management in Communication Networks, PhD Thesis, Columbia University, 1990.

4. Comer, D.E., *Internetworking with TCP/IP*, vol. **1**, Prentice Hall, Inc., Englewoods Cliffs, NJ, 1991.
5. DEC OSF/1, Configuring Your Network Software, *Digital Equipment Corporation*, 1993.
6. El Fattah, Y. and Dechter, R., Empirical Evaluation of Diagnosis as Optimization in Constraint Networks, *Working Papers of The Third International Workshop on Principles of Diagnosis (DX-92)*, (1992).
7. Freuder, E.C. and Mackworth, A.K., Special Volume, Constraint-Based Reasoning, *Artificial Intelligence*, **58**, 1992.
8. Freuder, E.C. and Wallace, R.J., Partial Constraint Satisfaction, *Artificial Intelligence*, **58**, 21–71, 1992.
9. Ghedamsi, A., von Bochmann, G. and Dssouli, R., Diagnosing multiple faults in finite state machines, Technical Report, Dept. d'IRO, Universite de Montreal, Canada, January 1993.
10. Hamscher, W., Console, L. and de Kleer, J., editors, *Readings in Model-based Diagnosis*, Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1992.
11. Hunt, C., *TCP/IP Network Administration*, O'Reilly & Associates, Inc., Sebastopol, CA, 1994.
12. ISO, ISO Open Systems Interconnection - Basic Reference Model, Second Edition, *ISO/TC 97/SC 16(ISO CD 7498-1)*, 1992.
13. Lin, Y.J. and Wuu, G., A constraint approach for temporal intervals in the analysis of timed transitions, *Protocol Specification, Testing and Verification*, **XI**, 215–230, 1991.
14. Miller, M.A., *Troubleshooting TCP/IP*, M&T Books, 1993.
15. Mittal, S. and Falkenhainer, B., Dynamic Constraint Satisfaction Problems, *Proceedings of the Eighth National Conference on Artificial Intelligence (AAAI-90)*, 25-32, 1990.
16. Mittal, S., Reasoning about Resource Constraints in Configuration Tasks, *SSL Technical Report, XEROX Park*, 1990.
17. Mittal, S. and Frayman, F., Towards a Generic Model of Configuration Tasks, *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, **2**, 1395–1401, 1989.
18. Mockapetris, P., Domain Names - Concepts and Facilities, *Request For Comments 1034*, 1987.
19. Postel, J., File Transfer Protocol, *Request For Comments 959*, ISI, October 1985.
20. Riese, M., Model-based Diagnosis of Communication Protocols, PhD Thesis, Swiss Federal Institute of Technology, Lausanne, 1993.
21. Wang, C.J. and Liu, M.T., A test suite generation method for extended finite state machines using axiomatic semantics approach, *Protocol Specification, Testing and Verification*, **XII**, 29–43, 1992.

This article was processed using the LaTeX macro package with LLNCS style