Troubleshooting with human-readable automated reasoning

Alva L. Couch
Tufts University

Mark Burgess
Oslo University College
and Cfengine AS

Abstract

In troubleshooting a complex system, hidden dependencies manifest in unexpected ways. We present a methodology for uncovering dependencies between behavior and configuration by exploiting what we call "weak transitive relationships" in the architecture of a system. The user specifies known architectural relationships between components, plus a set of inference rules for discovering new ones. A software system uses these to infer new relationships and suggest culprits that might cause a specific behavior. This serves both as a memory aid and to quickly enumerate potential causes of symptoms. Architectural descriptions, including selected data from Configuration Management Databases (CMDB) contain most of the information needed to perform this analysis. Thus the user can obtain valuable information from such a database with little effort.

1 Introduction

Troubleshooting is about linking symptoms with causes. The speed of troubleshooting depends upon how quickly one can do that, as well as how complete the list of potential causes can be made. It can further be enhanced so that more frequent causes are checked first. In a very complex system, it can be laborious to make a list all of the potential causes of a behavior.

In this paper, we present a method for deriving a description of causal relationships from a description of system knowledge. This method maps symptoms to possible causes via a methodology that we call "weak transitivity". Architectural facts and logical inference rules describe relationships between architecture and causation in a knowledge network. So, while architecture might vary, inference rules, delimiting meanings of relationships, are invariant and reusable. One can use these rules to efficiently reason about potential causes and to eliminate options incrementally as troubleshooting progresses. The system's logic and reasoning are straightforward, simple to understand, and scalable to arbitrarily large networks.

The key contributions of this work include:

1. An exterior ("black box") model of the meaning of relationships between architectural components, that permits logical inference based on incomplete or partial information.

2. The ability to exploit existing knowledge – e.g in Configuration Management Databases – to aid in the troubleshooting process.

3. The ability to generate a human-readable explanation of the possibly subtle relationships between components.

4. A set of useful, reusable classes and relationships along with rules that define their meanings.

2 Background

Our work arose from ideas for and against the use of logical reasoning in system administration[4, 9], but we approach the problem of applying logic to system administration from a new angle based on knowledge representation, specifically Topic Maps[23, 24]. In using topic maps to index documentation, we found that a particular way of thinking about the map led

to more efficient use of documentation. If we view the map as a set of *links* between topics, it is easy to get lost in the map, while if we view a map as a set of chains of *reasoning*, the relationships become clearer and the map becomes more useful[7]. The same kind of reasoning that can be used to understand documentation can be utilized to understand complex *systems*. This paper applies our approach to the specific task of troubleshooting, which is – at its core – a problem of understanding and coping with what is known and unknown.

There are plenty of other approaches to troubleshooting[25]. Snitch[17] applies a maximum-entropy approach to creating dynamic decision trees for troubleshooting support, using a probabilistic model inferred from practice. Snitch is related to "revealed causal modeling"[18, 19], which also attempts to measure causality as a set of probabilities of relationships. Troubleshooting has an intimate relationship with cost of operations[12], which justifies use of decision trees and other probabilistic tools to minimize cost and maximize value. The Maelstrom approach[8] exploits self-organization in troubleshooting to re-organize the process based upon hidden precedences. STRIDER[26] employs knowledge of behavior of similar hosts and Windows registries to infer possible trouble points. Outside the system administration domain, SASCO[15] guides troubleshooting by heuristics, using what it calls a "greedy approach" to pick most likely paths to a solution.

There are several differences between our work and these prior approaches to troubleshooting. We base our troubleshooting upon an incomplete description of the *architecture* of the system under test, rather than statistical information about likelihood. We use architectural reasoning to infer the nature of dependencies in the system, and use those inferences to guide troubleshooting. This leads to a synergy between the accuracy of the description and effectiveness of troubleshooting, which leads in turn to increasing accuracy of the architectural information as it is revised to reflect observations. The net result is that we show how to apply something we already need to have – a global map of the architecture – to the troubleshooting process.

2.1 Formal reasoning

While it is certainly a kind of formal reasoning, this work is difficult to place in the context of other approaches to formal reasoning. It is a form of logical abduction[13, 16, 20] that explains connections between entities. Very complex systems have been built to reason using abduction, but none of these is guaranteed to output an easily understandable sequence of logical dependencies. Our method has its roots in using logic programming for configuration management[9], but also takes inspiration from methods used to manipulate topic maps in library science[23, 24], and is closely related to ontological reasoning in the semantic web. Unlike ontological reasoning, which attempts to match concepts based upon their interaction with others, we concentrate on inferring relationships between individual entities, based upon facts and rules that describe an architecture.

Our methods are somewhat removed from traditional approaches to logical inference and computer logic. First, we sidestep the difficult problem of reasoning with modal logic, by encoding modality into our relationships. A "modal logic" includes the ability to distinguish modal facts in English, e.g., "X might affect Y" from non-modal facts, e.g., "X affects Y." Instead of modeling modality, we incorporate all modality into our relationships, which makes our rules for relationships somewhat more complex, but also reusable and perhaps easier to compute.

2.2 Information modeling

Our work includes a limited form of information modeling as proposed by Parsons[21]. However, our notation escapes what Parsons calls the "tyranny of classification"[22] in which an instance *must* be a member of some class. We escape that tyranny by only partially defining such classifications, and leaving what is unknown out of the data specification. Likewise, our data are much simpler in structure from that in the Shared Information and Data model(SID)[14], mostly due to lack of structure (or even the need for structure) in our approach.

2.3 Knowledge management

Our problem is a sub-problem of the larger issue of knowledge management for complex networks. Knowledge management is a key challenge of the coming decade. The technologies and tools for system administration and configuration management have all progressed to the point where the main difficulty lies in the knowledge required to integrate them to produce a seamless IT infrastructure. With many of the basic problems of system administration essentially solved, a major system administration concern of the next decade will be loss of business continuity, due to inability to maintain and utilize appropriate systems knowledge. For example, when system administrators are fired or leave, the business can suffer from lack of knowledge of what they did, resulting in increased downtime, risk, and cost. It costs real money for a new system administrator to learn what his or her predecessors did. Knowledge and understanding of system complexity are also major limitations to system growth (scalability).

Cfengine was recently redesigned with knowledge in mind, using a model of "promises"[1, 4, 5, 6] that separates the intentions of system components from the mechanism by which they achieve those intentions. Promises combine clearly defined goals with self-documenting statements that have an associative structure. From there, it is a small step to create an associative meta-model (semantic web) of promises, which can be integrated with any other kind of semantically annotated documentation. Such a knowledge model can be used not only for searching for relevant information, but also for reasoning and for encoding expertise. Expert systems have been discussed many times before, but they are usually data-intensive and expensive to maintain. Here, we present a mechanism that is both cheaper and is designed to work for humans rather than to replace them. Most important, it arises naturally from the act of managing systems with Cfengine and requires no separate data collection.

2.4 Configuration Management Databases (CMDB)

Configuration Management Databases (CMDB), as defined by the IT Infrastructure Library (ITIL), gather system data, usually in a brute-force taxonomic form. Common data models in use include the Common Information Model (CIM) and the Shared Information and Data Model (SID). These concentrate on configuration *data* of specific hosts, while their meanings and inter-relationships are assumed to be entirely implicit in the taxonomy. The problem with this (and all hierarchical classifications) is that new information can only be introduced by expanding the model itself.

Our technology was developed specifically for Cfengine and its `cf-know` utility (where the required architectural model is available), though we describe the techniques we use more generally here. The lesson from Cfengine is that meta-models with weak constraints avoid many of the pitfalls of 'Object Oriented' hierarchical classification. The techniques can be used with any kind of configuration management database, provided that one can mine appropriate kinds of relationships from it.

3 A motivating example

Using architectural knowledge for troubleshooting might be a counter-intuitive idea, so here is a simple example. Suppose we have a very simple network with a fileserver '`host01`', a DNS server '`host02`', and a client workstation '`host03`'. We might code the relationships between these hosts as a set of abstract "sentences", like:

```
host01 | is an instance of | file server
file server | provides | file service
host02 | is an instance of | dns server
dns server | provides | dns service
host03 | is an instance of | client workstation
client workstation | requires | file service
client workstation | requires | dns service
```

Each sentence has a subject, a verb, and an object separated by vertical bars (|). Sentences are preparsed into subject, verb, and object *by the user*; no natural language parsing is employed. We call each

such sentence a *fact*[1].

From the base facts above, we can intuit several other facts, including:

```
host01 | provides | file service
host02 | provides | dns service
host03 | requires | file service
host03 | requires | dns service
host03 | might depend upon | host01
host03 | might depend upon | host02
```

The last two are subtle: the fact that a host provides something does not mean that it provides it to everyone who requires it.

Suppose that something goes wrong with this network, e.g., `host03` stops responding. The main problem in troubleshooting is to enumerate the entities that can cause the symptoms, rule out causes, and thus determine *where to look* for problems. Obviously, one symptom is that `host03` is broken, which according to the above can be due to a problem with `host03`, a problem with *host01*, a problem with `host02`, or a problem with the network connecting the hosts. If we know more, e.g., that the network is functional but that `host03` DNS service is broken, then this rules out `host01` and points to either `host02` or `host03` as potentially problematic. If we know as well that DNS is functional and `host03`'s configuration for DNS is correct, this points toward `host02`. In other words, the more we know, the more we can eliminate and the narrower the sieve of options becomes.

What our system does is to suggest possibilities in order of approximate likelihood, based upon a description of architecture. For example, in the above it would first report the dependencies upon `host01` and `host02`, which are the "closest" possible causes according to a notion of distance based upon the number of logical inferences required to connect two entities. Then, for each possibility, it can "explain" the relationship between a probable cause and the symptom, all from a description of architecture.

In this trivial case, one can easily do this by hand. With systems of thousands of components, however, the problem becomes more complex. In this paper, we describe a mechanism whereby one can reason about very complex architectures and obtain explanations of complex dependencies between subsystems. We verify our thinking via a simple prototype that serves as a proof of concept. In describing our ideas, we utilize the notation of the prototype, to encourage system administrators to try it out with their data and see what it can do for them.

4 Entities and relationships

The key to our solution is a description (cached as a database) describing the *architecture* of the underlying system. The role of this description is to serve as a model of locations and interactions. For this, we appeal to a very old idea: entity-relationship modeling[2]. We describe the network as a graph of named entities and relationships, either manually or by mining the configuration.

Entities in the network are named by strings and can be named at any level, including subnet, host, component, or even software package. Kinds of entities include:

1. physical machines, e.g., '`host01`'.
2. software, e.g., '`RHEL5`', '`Linux`'.
3. services, e.g., '`LDAP`', '`SMTP`', '`HTTP`'.
4. classes of physical items, e.g., '`webserver`', '`mailserver`'.

An entity is a noun whose meaning does not change over time. Nouns can represent classes of things, e.g. '`client workstation`'.

Relationships can be anything, including:

1. Dependencies, including '`requires`' and '`provides`'.
2. Containment, including '`is a part of`', '`is an instance of`'.

[1] Functionally, these are just like facts in the logic programming language Prolog, where our fact '`client workstation | requires | file service`' becomes the Prolog fact `requires(client_workstation, file_service)`.

[2] We refer specifically to the ER-diagrams utilized in Software Engineering, as opposed to those utilized in database theory. The former describe interactions, while the latter describe functional dependencies.

3. Causality, including '`determines`', '`influences`'.

4. Connectivity, including '`connected to`'.

5. Intent, including '`promises`', '`uses`'.

While entities are nouns, relationships are (usually) verbs. Any invariant relationship can be documented. Verbs can also represent classes of relationships, e.g., '`determines`', which allows many different *kinds* of determination.

Most relationships are directional, i.e., if '`A | is a part of | B`' one cannot conclude that '`B | is a part of | A`', any more than "A is a part of B" would imply that "B is a part of A" in English. However, every relationship corresponds to a unique *inverse relationship* that is simply another predefined formal symbol. If '`A | is a part of | B`', then '`B | has part | A`'. The formal symbol '`has part`' is *defined* as the "inverse" of the formal symbol '`is a part of`'.

5 Relationship to topic maps

One can also think of entities as "topics" and relationships as "associations" between topics in a topic map[24, 23]. This is a kind of generalized ER-model utilized usually in library science[3]. Unlike our simplified ER-model, a topic map describes relationships between three kinds of entities[23]:

1. *Topics* (entities) are analogous to entries in an index of a book.

2. *Associations* (relationships) are analogous to "See also" in a book index.

3. *Occurrences* are are analogous to page numbers in an index, and specify "where" a topic is mentioned.

[3]In this paper, we will concentrate on a simple application of the idea, and not a broader view. While what we do here *can* be utilized with a variety of kinds of data, we concentrate specifically on troubleshooting data and avoid more general problem statements for clarity.

While this work was inspired by initial work in topic maps, the results presented here are more broadly applicable to any ER-model.

The most important thing we draw from topic maps is the *semantics* of our representations. Our ER-diagrams, like topic maps, are intended to *define* entities through their relationships with other entities. Throughout this paper, we will make design decisions that preserve "definition-like" qualities for both entities and relationships. Notably:

1. Entities are static and do not change over time (from the point of view of the reasoning system, inside the formal model).

2. Relationships are static and do not change over time.

3. Definitions are additive and define *facets* of a thing. The total definition of a thing is the union of partial definitions (just as in a dictionary).

Our definition of inverses as verb phrases is consistent with the Cfengine-3 notion of inverse relationships, but differs from the more refined notion of inverses in topic maps. In a topic map, a relationship is a *noun phrase*, and the meanings of sides of the relationship are clarified via what are called "roles". For example, our statement '`cat food`' '`is manufactured by`' '`pet food companies`' would be written in a topic map as "cat food" in role of "product" has relationship "manufacture" to "pet food companies" in role of "manufacturer". We do not need this extra complexity, so we sidestep it. What we lose from this is that our prototype is only compatible with "subject-verb-object" (SVO) natural languages (e.g., like English, French, etc.), as opposed to "subject-object-verb" (SOV) languages (e.g., Arabic, Japanese, etc.). The topic map mechanism handles both SVO and SOV languages, by translating relationships into foreign languages *after* processing.

6 Facts

The first step in utilizing our system is to create (or transform) an appropriate database of suitable facts.

Each fact is a subject-verb-object triple, where subject and object are system entities (or classes), and the verb indicates some kind of relationship between the two entities. Common kinds of facts include class membership, e.g.,

```
couch1 | is an instance of | client workstation
```

class descriptions, e.g.,

```
client workstation | requires | file service
```

and ownership, e.g.,

```
couch1 | is owned by | Alva L. Couch
```

There is no checking as to whether facts make sense in English. The system trusts the user to use relationships that are transitive verbs, and subject and object that are nouns. There is no natural language processing at all in the system. Subject, verb, and object in the fact are syntactic tokens, and nothing more.

6.1 Coding and avoiding hierarchy

Note that the way we specify facts looks very similar to object-oriented modeling, but there is an important difference. Our encoding method is non-hierarchical, in the sense that there is no need to place each host into a hierarchy of relationships. One can do this when convenient, but it is not necessary to the reasoning method. Thus one need not become subject to the "tyranny of classification", in which hierarchy impedes information encoding[22]. Instead, one can freely classify objects into *several* convenient hierarchies, without contradiction. A machine can be a kind of server, a member of an ownership hierarchy, and a kind of client, with no confusion.

Hierarchy is not absent from our system; it is simply *not essential.* Complex entities with many parts are easily modeled via part and subclass relationships, e.g.,

```
dns server | has part | dns local zone information
dns server | has part | dns configuration file
dns server | is an instance of | server
```

with the obvious meanings. Users and privilege can be modeled straightforwardly by thinking of the user as a primary key:

```
Alva | refers to person | Alva L. Couch
Alva | uses shell | /bin/bash
Alva | administers | couch1
Alva | administers | couch2
```

to describe an entity '`Alva`' who administers two machines '`couch1`' and '`couch2`'.

As in the preceding example, one describes multiple relationships by listing instances:

```
Mark | administers | couch1
Alva | administers | couch1
```

means that *both* administer '`couch1`'. Sets of facts are treated as if all are true, i.e., listing two facts implicitly connects them with logical 'and'.

There is no equivalent to logical 'or' in the calculus, nor is there any equivalent to negation. To express that something is one thing or another, one can construct a (synthetic) class '`admin1`' with more than one instance:

```
Mark | is an instance of | admin1
Alva | is an instance of | admin1
admin1 | administers | couch1
```

to denote that *some instance* of the class '`admin1`' administers '`couch1`'.

Note that when a class is used in a fact, an instance is *implied*; no class can "administer" anything. However, this form of disjunction is not exclusive and thus does not preclude that both '`Mark`' and '`Alva`' administer '`couch1`'.

6.2 Modal facts

In our reasoning system, there are very precise meanings of modal expressions in English such as '`X | can serve | Y`' or '`X | might serve | Y`'. The qualifier 'can' implies capability but not intent: '`X | can serve | Y`' means that `X` is capable of serving `Y` but not that `X` is actually serving `Y`. The qualifier 'might' means that there is some (as yet unknown) possibility of a thing. If we say '`X | might serve | Y`', this means that in some *worlds*, `X` serves `Y` and in others, `X` is not known to serve `Y`. These are strength indicators for one's confidence that something is true: 'might' is stronger than 'can'. Neither of these is a temporal distinction; if something might serve something else, it still does or does not serve it, i.e., either '`X |`

`serves | Y`' is a fact, or not. The modal fact encodes the possibility that the non-modal fact is present. Later we will see that modal constructions have a complex interaction with class membership ('`is an instance of`') and structural ('`is a part of`') relationships.

6.3 Pitfalls in declaring facts

The main trap in representing a fact is to represent "too much", so that the implications of a fact far exceed what is intended. Representing "too much" costs the administrator time in sifting through impossible alternatives, while representing "too little" does not depict valid alternatives. Thus, the best practical advice is "when in doubt, specify too little.".

Another way to say this is that one should adopt a "maximum entropy principle" that what is not known for sure is not considered to be known at all.

For example, suppose you do not really know that a client workstation utilizes a specific file server. It would be bad to declare that it uses something that it might not, but fairly harmless to declare that it uses *some* file server, identity unknown. The former will misdirect the reasoning system, while the latter will point out to the reasoning system that this particular facet of configuration is unknown, leading to possibility rather than hard fact. This is what happened in the inferences in the first example, where the relationship '`might depend upon`' indicates that uncertainty.

Another pitfall of encoding facts might best be called the "tyranny of naming". A system entity is often best described by its attributes rather than its name. The name of an object is – at best – nothing more than a (hopefully) unique key. Obviously, it is very bad to use the same name for two distinct things. It might be best, therefore, to use automatically generated unique names for entities, e.g., '`id29394510`', and let attributes of the objects define their physical identities, e.g.,

```
id29394510 | has hostname | couch1
id29394510 | has manufacturer | dell
id29394510 | has serial | 000-123-4567
id29394510 | is owned by | Alva
```

The unique key '`id29394510`' need not be central to a query; one can ask the system what entities influence the (human) '`Alva`', and it can respond with, e.g., hostnames.

A third pitfall of encoding facts is that – because of the simplicity of our representation – relationships often imply the types of their arguments. For example, if one has the fact:

```
host01 | provides | dns service
```

then it is implicit in the relationship '`provides`' that '`host01`' is either a machine or a class of machines. Saying, e.g., that:

```
Alva L. Couch | provides | dns service
```

is thus made somewhat nonsensical – a person can't be a machine or instance of a machine.

In topic maps, this ambiguity is resolved via the concept of *roles*, which determine the types of the subject and object of a relationship. Thus, notating roles as subscripts, one might write:

```
host01 | machine provides service | dns service
```

to encode the fact that '`host01`' is an instance of the generic class '`machine`' and '`dns service`' is an instance of the generic class '`service`'. In this case, the relationship between '`host01`' and '`dns service`' is the ternary symbol $\texttt{machine}^{\texttt{provides}}_{\texttt{service}}$, where roles are listed on the side to which they apply. In the interest of simplicity, for this paper, roles will remain implicit, but in general, roles can be useful to disambiguate between relationships that are, in fact, different: e.g., $\texttt{machine}^{\texttt{provides}}_{\texttt{service}}$ versus $\texttt{person}^{\texttt{provides}}_{\texttt{service}}$.

7 Rules

We reason about troubleshooting using a simple calculus of facts and rules that is inspired by – but somewhat different from – ontological reasoning in the semantic web. During ontological reasoning, one connects two entities by looking at how they interact with other entities. Two entities are "similar" if they interact with nearly the same other entities. By contrast, our rules do not concern similarity between

entities, but instead derive relationships from relationships, without considering how entities are similar or dissimilar. Rules suggest new facts in several ways, including canonicalization, inverse relationships, transitive relationships, and implications.

7.1 Canonicalization

The purpose of canonicalization is to both save typing and ensure that representations of facts are sufficiently precise to be useful. The relationship '`is a`' is ambiguous; `X | is a | Y` could mean that X is an instance of Y, or that X is a kind of Y. The canonicalization:

```
is a => is an instance of
```

disambiguates between these two alternatives (and more). Canonicalizations are always denoted by "=>", and allow one to utilize a shorthand when writing rules that is expanded later. In the prototype implementation, we employ the following canonicalizations:

```
is a superclass of => has subclass
has superclass => is a subclass of
```

to ensure that we only talk about subclass relationships rather than the equivalent superclass relationships. This is so all class relationships will be comparable.

7.2 Inverses

Inverses allow one to reverse a relationship so that the object switches positions with the subject. The *inverse rule*

```
is an instance of <> has instance
```

means that for every X and Y, if '`X | is an instance of | Y`', then '`Y | has instance | X`' (and vice-versa). The inverse for a relationship is the English phrase that – in English – represents the reversed relationship. Inverses are syntactic, and not semantic. They are always defined, and never inferred.

A few relationships are self-inverse, i.e., '`is a peer of <> is a peer of`', because '`A | is a peer of | B`' exactly when '`B | is a peer of | A`'. Most inverses are simply other ways of stating the same relationship, such as '`is an instance of <> has instance`', which means that '`A | is an instance of | B`' exactly when '`B | has instance | A`'.

The meaning of an inverse in English is incidental to its use. E.g., if you define '`foo <> bar`', then these relationships are inverses, regardless of what they might mean in English; this rule means that if '`Alva | foo | Mark`', then '`Mark | bar | Alva`'.

7.3 Weak transitive rules

Weak transitive rules make connections between previously unconnected objects.

In mathematics, a *transitive relation* is a set of ordered pairs R where for any A, B, and C, if $(A, B) \in R$ and $(B, C) \in R$, then $(A, C) \in R$. In our context, a transitive relation is represented by a verb phrase `R` where for any nouns `A`, `B`, `C`, if `A | R| B` and `B | R | C`, then `A | R | C`. For example, '`is a part of`' is transitive: if A is a part of B, and B is a part of C, then A is always a part of C. Examples of some transitive relations are shown in Table 1.

Each of these relations corresponds to a transitive *rule* in our reasoning system. The relations in the table correspond to the rules

```
is larger than ^ is larger than ^ is larger than
is caused by ^ is caused by ^ is caused by
is the same as ^ is the same as ^ is the same as
depends upon  ^ depends upon  ^ depends upon
is a part of  ^ is a part of  ^ is a part of
is the same as ^ is the same as ^ is the same as
```

where "^" delimits relationships.

However, in our system, there are rules that look somewhat like the former, but whose antecedent and consequent relationships differ from one another. We call these *weak transitive rules*, because they look somewhat like transitive rules but are not. For example, if A is an instance of B and B provides C, then A provides C, meaning that if something is a member of a class that does something, the instance does it too. Some examples of weak transitive rules are listed in Table 2. We notate weak transitive rules in the same way as transitive rules; the rules in the table are notated as

Fact 1	Fact 2	Implies...
A is larger than B	B is larger than C	A is larger than C
A is caused by B	B is caused by C	A is caused by C
A is the same as B	B is the same as C	A is the same as C
A depends upon B	B depends upon C	A depends upon C
A is a part of B	B is a part of C	A is a part of C
A is the same as B	B is the same as C	A is the same as C

Table 1: Transitive relationships correspond to transitive rules.

Fact 1	Fact 2	Implies...
A is an instance of B	B provides C	A provides C
A is an instance of B	B requires C	A requires C
A is larger than B	B might be larger than C	A might be larger than C
A might be larger than B	B is larger than C	A might be larger than C
A depends upon B	B might be influenced by C	A might be influenced by C

Table 2: Weak transitive rules look like transitive rules except that antecedents and consequent differ in some way.

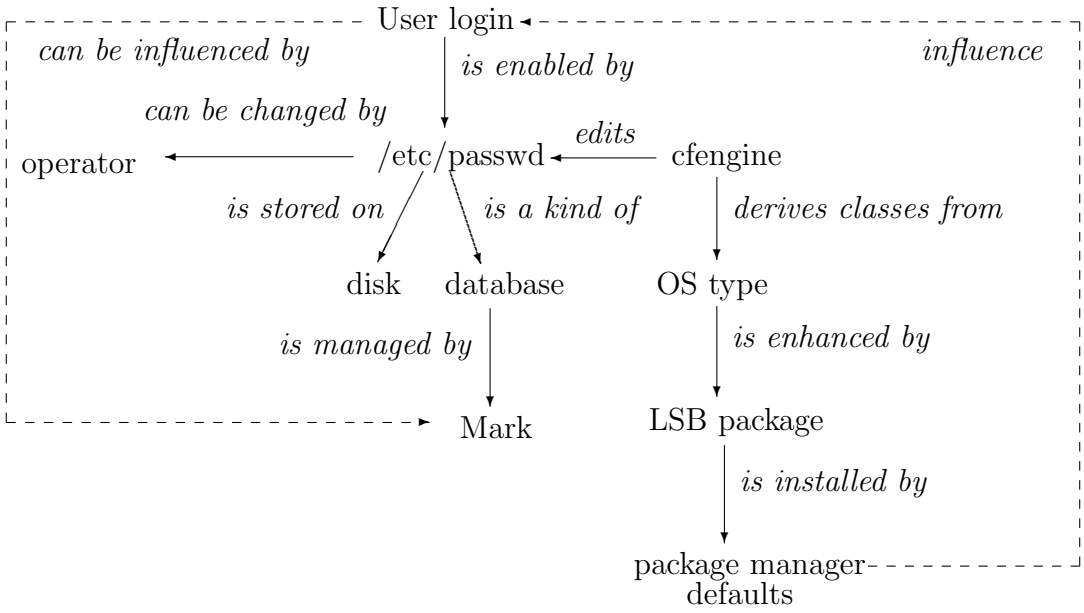

Figure 1: One useful depiction of architecture is a graph in which nodes are entities and arrows represent relationships. Base facts are depicted as solid lines, while two inferred facts are depicted as dashed lines.

```
is an instance of ^ provides ^ provides
is an instance of ^ requires ^ requires
is larger than ^ might be larger than ^ might be
    larger than
might be larger than ^ is larger than ^ might be
    larger than
depends upon ^ might be influenced by ^ might be
    influenced by
```

The point of weak transitive rules is to allow us to codify all ways in which two entities can be connected to one another. Each rule provides one form of connection, and these are the only rules in our system that make new connections.

While transitive rules often result in strong connections (e.g., '`determines`'), weak transitive rules often result in weaker connections (e.g., '`might influence`') that say less about the relationship between the two entities. The point of weak connections is that, even when strong connections do not exist, weaker relationships can guide humans in finding problems. Weak transitivity, as we define it here, offers a simple and measured approach for enumerating possibilities.

7.4 Implications

Implication rules allow one to change the level of abstraction at which reasoning occurs. The *implication rule*

```
provides -> determines
```

means that for every pair of entities '`X`' and '`Y`', if '`X | provides | Y`' then '`X | determines | Y`'. The purpose of implication in our system is to allow one to raise the level of abstraction to a level at which reasoning can occur. If '`Z -> W`', then Z is more specific than W, and W is more abstract (generic) than Z. Specific facts may have no obvious inter-relationship, while their generic equivalents may be obviously related.

For example, consider the facts:

```
host01 | is a file server for | host02
host02 | provides | print service
```

On the surface, these do not have any relationship to each other. But if we translate to a higher level of abstraction via the implications:

```
is a file server for -> influences
provides -> influences
```

then we get the higher-level facts

```
host01 | influences | host02
host02 | influences | print service
```

Then, by the transitive rule:

```
influences ^ influences ^ influences
```

we obtain the new fact

```
host01 | influences | print service
```

which might be quite important to know. In this example – and many others – raising the level of abstraction exposes relationships that are not apparent at lower levels.

7.5 An example of reasoning

Consider the example in Figure 1. A user is unable to log on to a given host, so a diagnostician points the prototype at the entity '`User login`'. The prototype invokes our algorithm to enumerate possibilities. These possibilities are relationships between entities, and not obviously anything that can be logically connected with faults. The human user must evaluate the possibilities.

For instance, if '`User login`' is enabled by the file '`/etc/passwd`', then it '`is influenced by`' it. If '`/etc/passwd`' can be changed by an operator, then it '`can be influenced by`' the operator. If '`/etc/passwd`' is stored on the '`disk`', then it '`is influenced by`' the disk. If '`/etc/passwd`' is a kind of '`database`' and databases are managed by '`Mark`', then '`/etc/passwd | can be influenced by | Mark`'.

But often, more subtle and hidden connections are the real cause of the problem. Here is a problem we have experienced in real practice. A possible but less than obvious cause of a missing user entry in '`/etc/passwd`' is that the file is being managed by an agent (like Cfengine), whose policy applies only to a certain operating system type. That operating system type is only detected in the prescribed manner if the package '`Linux Standard Base`' (LSB) is installed. This in turn depends on the default settings for the package manager in use. In other words, the default settings of the package manager *influence* user login.

What we see in this example is the power of lateral thinking. The system generates alternatives and the administrator rules out each one in turn. The system does not perform logical elimination to find the cause of a fault, but rather the opposite: it enumerates possibilities the administrator may not have considered.

7.6 Rules as shorthands

One purpose of rules in our system is to shorten notation and to allow automatic inference of related facts. We could– in principle – simply enumerate all facts, but this would be a laborious process. One rule suffices as a substitute for writing down many facts.

For example, suppose that there are entities '`LDAP`', '`login privileges`', and '`shell access`', where

```
LDAP ^ can determine ^ login privileges
login privileges ^ can determine ^ shell access
```

Note that these relationships describe potential for interaction, rather than assurance of interaction.

Implications allow us to avoid writing down obvious outcomes. The rather obvious rule

```
can determine -> might determine
```

denotes that the *capability* to do a thing is necessary in order for the *possibility* to do a thing. This rule means that we do not have to write down the facts:

```
LDAP | might determine | login privileges
login privileges | might determine | shell access
```

because these are implied by the facts above and the implication.

Likewise, if there is a transitive rule

```
can determine ^ can determine ^ can determine
```

then we do not have to write down the fact

```
LDAP | can determine | shell access
```

because the latter is a result of that rule and the base facts above.

Rules can also *interact* with each other to produce new rules. The implication

```
can determine -> might determine
```

and the transitive rule

```
can determine ^ can determine ^ can determine
```

together imply the rule

```
can determine ^ can determine ^ might determine
```

because *possibility is weaker than capability.* The rule still applies if the consequent of the rule is weakened.

Moreover, if we have the obvious implication and transitive laws

```
determines -> can determine
can determine ^ can determine ^ can determine
```

then we also can infer the *rules*

```
determines ^ determines ^ can determine
can determine ^ determines ^ can determine
determines ^ can determine ^ can determine
```

because the rule still applies if either of the antecedents are strengthened, and '`determines`' is stronger than '`can determine`'. Also, from

```
determines -> can determine
determines ^ determines ^ determines
```

we can infer that

```
determines ^ determines ^ can determine
```

In general, any rule remains valid if we *strengthen the antecedents* and/or *weaken the consequent.* This is how the prototype actually works internally, and is part of the reason it is efficient.

7.7 Rules as meaning

Another unique aspect of our system is how meaning is imparted to symbols. In most logical systems there is some external model that defines what symbols mean. In our system, *the meaning is the rules.* The interactions between the relationship '`influences`' and other relationships *comprise* its meaning, and two different tokens (e.g., '`influences`' and '`coerces`') are identical whenever their interactions with the other tokens are the same. In other words, ontological equivalence between relationships implies that the relationships have the exact same meaning (with respect to all other relationships considered in the rules).

To understand this (rather subtle) idea, consider the rules

```
determines -> influences
determines -> can determine
can determine -> might determine
influences -> can influence
can influence -> might influence
influences ^ is a part of ^ influences
is a part of ^ influences ^ influences
determines ^ is a part of ^ influences
is a part of ^ determines ^ determines
influences ^ is an instance of ^ might influence
is an instance of ^ influences ^ influences
determines ^ is an instance of ^ might determine
is an instance of ^ determines ^ determines
```

These rules – in a nutshell – encode the principal semantic differences between '`influences`' and '`determines`' with respect to '`is a part of`' and '`is an instance of`'. Note that if one influences an instance of a thing, then one *might influence* all instances (the containing class), or not. If one determines a thing, then one determines its part, but if one determines a thing that is a part of another, one only influences the larger thing. We consider this interaction to be part of the *definition* of the relationships '`determines`' and '`influences`'.

7.8 Classes and structures

The rules for classes and structures deserve special comment. As in object-oriented modeling, a *class* of things shares some common attributes and has instances that have those attributes. Likewise, a *structure* has parts.

For classes, note that

```
is an instance of ^ has attribute ^ has attribute
```

is almost the *definition* of a class. But, perhaps counter-intuitively

```
has attribute ^ is an instance of ^ might have
    attribute
```

because the existence of an attribute in an instance does not mean that it is present in all instances (and thus the class). An instance might be also an instance of a subclass.

Causal relationships have some subtleties. Straightforwardly,

```
is an instance of ^ is determined by ^ is
    determined by
```

because determining all of a class determines its instances. But

```
is determined by ^ is an instance of ^ might be
    determined by
```

because the fact that an instance determines something does not mean that *every* instance determines it.

For structures, note that

```
determines ^ has part ^ determines
```

because if one determines a thing, one determines all parts. But rather obviously,

```
has part ^ determines ^ influences
```

because determining a part does not implicitly determine the whole thing. Again, some subtleties arise:

```
influences ^ has part ^ might influence
has part ^ influences ^ might influence
```

because if something is a part of something larger, and we influence the whole thing, we might or might not touch a specific part. These rules might be considered the definition of '`has part`'.

8 Philosophical concerns

In using our system, several strongly held philosophical decisions become immediately obvious. We designed the system around an open model of knowledge, in the sense that no model is considered to be complete. We also designed the inference system so that knowledge is convergent, in the sense that multiple rounds of inference converge to a fixed-point knowledge base in a finite number of iterations. These decisions give the reasoning system both speed and scalability, but also match the fundamental philosophy of Cfengine upon which the system is based.

8.1 Open knowledge

There are two ways of conceptualizing a knowledge model. A *closed-world model* attempts to describe everything about a system, so that *facts that are absent are assumed to be false.* In an *open-world model*[10], facts describe only what is known, and leave other facets to be described later. When a fact is absent, this does not mean that it is false, but simply that *it is not known to be true* (yet). It might become known

to be true in the future, or not. In other words, open world models are ambiguous about whether the lack of a fact implies that it is false.

Like Cfengine, we adhere to an open-world philosophy. We never assume that our knowledge model is complete (or 'closed'), and err on the side of trying not to claim anything that is false. This makes it extremely easy to add information later, once it is known, while leveraging what is known in the meantime. Incompleteness of the architectural model does not hamper its use if we remember that it is incomplete.

8.2 Convergent inference

Another concept we borrow from Cfengine is the notion of convergence[2, 11, 9]. We think of the inference system as creating new facts from old facts, and new rules from old rules. An inference system is convergent if – by some finite number of applications of rules – it achieves a *fixed point state* in which no further operations add new facts or rules[3, 6].

The reason for this philosophical stance is computational. This will allow us (in the future) to code the inferences on a cloud at massive scale, because we can compute resultant facts in advance and then use Map/Reduce to find them[7]. This allows us to turn a logic problem into a database search problem, greatly simplifying implementation.

9 Queries

In the process of troubleshooting, the reasoning system provides guidance as to possibilities by answering several kinds of questions. These questions include what entities are potentially related to a subsystem, and precisely how two given entities are related to one another.

9.1 What are nearby entities?

In a complex system, on average, the most closely related entities to a symptom are most likely to contain the problem. Given an entity or set of entities with symptoms, the system can list those entities with some connection to the set, either via facts or rules. The 'closest' entities are those with some direct connection via a fact or implied fact, while more 'distant' entities are connected via weak transitive laws. The distance between two entities (with respect to some target relationship) is the number of weak transitive laws applied to connect them, plus 1. Entities directly connected by a fact are distance 1 apart, and every application of a weak transitive law adds 1.

Our concept of distance depends upon adopting some target relationship as a goal. Typically, the relationship of interest is '`might influence`', for some very subtle reasons. First, the reasoning system becomes more powerful as the level of abstraction increases. The relationship '`might influence`' is the most abstract relationship that is useful in troubleshooting. While we might actually be more interested in '`determines`', few strong lines of determinism arise in a realistic set of facts. The relationship '`might influence`' has several more concrete versions, specified via the implications

```
determines -> influences
determines -> can determine
can determine -> might determine
influences -> can influence
can influence -> might influence
```

where 'can' implies 'might' because *capability precedes possibility.* Thus '`might influence`' is a "more abstract" relationship than any of '`determines`', '`can determine`', '`might determine`', '`influences`', and '`can influence`', simply because it is more general and applies to more pairs of entities.

Implications are not counted as distance, because all they do is to restate a fact in a *different and less specific language*, and do not change the nature of the fact. By contrast, weak transitive rules add new facts and connections that were never explicit before.

9.2 What is the connection?

The second kind of query explains the connection between two entities. This gives guidance to the troubleshooter trying to debug that connection.

A *story* is a human-readable explanation of why some relationship exists. One can think of it as a "mathematical proof" of the soundness of reasoning.

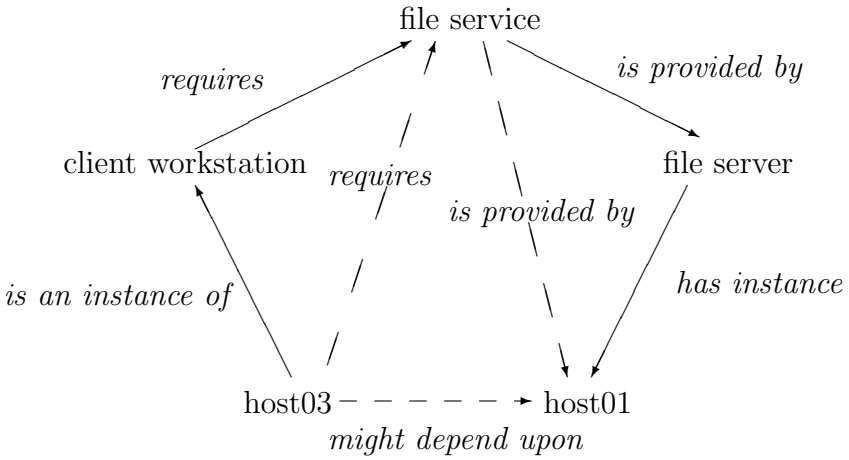

Figure 2: A chord diagram depicts entities in a story in a circle, while relationships are depicted as chord lines of the circle. Solid arrows represent facts, while inferred relationships are represented by dashed arrows. The story explains the dashed horizontal line at the bottom.

One key attribute of our system is its ability to generate easily readable stories.

As a really simple example, suppose we want to know the relationship between '`host01`' and '`host03`' in the initial example. The system utilizes the facts:

```
host01 | is an instance of | file server
file server | provides | file service
host03 | is an instance of | client workstation
client workstation | requires | file service
```

and the weak transitive rules:

```
is an instance of ^ requires ^ requires
is provided by ^ has instance ^ is provided by
requires ^ is provided by ^ might depend upon
```

to infer that:

```
host03 | might depend upon | host01
```

The difference between our system and other forms of logical reasoning is that we have crafted the system so that this inference, once discovered, can be *explained*. An explanation of a relationship is a linear chain of entities and relationships whose combination via rules results in the relationship in question, e.g.,

```
host03 | is an instance of
| client workstation | requires
| file service | provided by
| file server | has instance
| host01
```

We call such an explanation a *story* of the relationship between '`host03`' and '`host01`'. Due to the nature of our rules, every high-level inference corresponds to at least one story (with perhaps many alternatives).

In the previous example, we have avoided depicting one thing, which is the specific set of rule applications that led to the story. In the example, one cannot simply apply rules from top to bottom. The series of rule applications can be depicted in a *chord diagram* (Figure 2), in which the entities are depicted in a circle and the base facts (before reasoning) are depicted as solid lines. The dashed lines (which are all chords of the circle) indicate inferred relationships.

9.3 Lifting and grounding

The preceding example was one of the simplest forms of reasoning of which the system is capable. Often, more trouble must be taken to make reasoning possible and understandable. Architectural descriptions are often incomplete and specified at different levels of abstraction. To cope with this, our system utilizes implication to "lift" facts to a *common level of abstraction or generality* at which reasoning can occur, and then "grounds" that reasoning by expressing the

high-level abstract facts in terms of the low-level facts that were their basis.

Consider, e.g., the following quandary:

```
host02 | is an instance of | file server
host03 | is an instance of | client workstation
client workstation | requires | file server
```

What is the real relationship or dependency between '`host02`' and '`host03`'?

To answer this question, we must proceed to a higher level of abstraction:

```
requires -> is influenced by
```

after which the facts available also include:

```
client workstation | is influenced by | file server
```

and, using the rules

```
is an instance of ^ is influenced by ^ is
    influenced by
is influenced by ^ has instance ^ might be
    influenced by
```

we infer that

```
host03 | might be influenced by | host02
```

from which we infer the story that:

```
host02 | is an instance of
| client workstation | is influenced by
| file server | has instance
| host03
```

but *this is not good enough*. To complete the picture, we "ground" the lifted relationships by replacing them with the concrete relationships that are their subclasses:

```
host02 | is an instance of
| client workstation | requires
| file server | has instance
| host03
```

which "explains" the abstract reasoning in more concrete terms.

10 A prototype

We implemented a prototype reasoning system as a web-based troubleshooting aid. In a troubleshooting situation, a user inputs locations at which symptoms have occurred, and the reasoning system responds with a likely list of other locations that might be the source of the problem. Options are listed in order of inference distance within the reasoning system, i.e., how many transitive rules had to be applied; we have found that this *roughly* corresponds to the strength of coupling between entities. Clicking upon a candidate source "explains" its relationship with the symptoms as a linear chain of dependences. The prototype is written in Perl, and the facts and rules are specified in a text file, using the notation in our examples. The current prototype does everything online. No pre-computed state is kept between queries. Thus the prototype is limited to relatively small examples, e.g., at most a few hundred entities. By contrast, the algorithm itself can be run on clouds, and can scale to arbitrary input sizes.

There are several ways this technology can be used to solve common troubleshooting problems. It can be used to remember details that might be otherwise forgotten, to learn about a new system with which one is unfamiliar, or even to debug one's architectural description of a system. The system does not replace human thought, but rather, assures that known facts are not forgotten.

10.1 Remembering details

First, the system aids a troubleshooter in remembering details or dependencies that might be missed. If one selects a trouble source, the system can respond with those hosts, services, or other entities that might be interfering with that source. For example, inputting '`DNS`' to the system (with relationship '`can influence`') gives a list of things that might affect DNS, in order of distance from DNS.

10.2 Exploring legacy systems

Another typical use case is to learn about legacy systems. System administrators change jobs more frequently than we would like to admit. If a prior administrator has documented the architecture, the new administrator faced with a new system can utilize the data to learn what dependencies are, and to get a feel for how things are connected. For example,

one can input two hosts and look for the dependencies between them, or one host and look for the hosts upon which it depends.

10.3 Debugging architectural descriptions

A final and not-so-typical use of the system is to debug architectural descriptions by examining the consequences of those descriptions. This occurs naturally as a result of using the system. When a relationship is explained, the chain of reasoning is presented in terms of the input facts. If an inference is incorrect, the cause must be an invalid input fact, and these are shown for every inference.

11 Critique

This method is not a panacea. It requires careful coding of relationships in order to avoid erroneous conclusions and wasted time. The "inference distance" metric used to determine "most likely" causes could use some refinement. Clearly, there are many shades of '`influences`', from '`greatly influences`' to '`slightly influences`'. The current calculus does not account for shades of meaning.

11.1 Sensitivity to definitions

On a related note, the core causal relationships must be rather rigorously defined in order for the system to work well. Our system "defines" relationships via their interactions with others. Our rules in some sense embody the definitions of our relationships. One must understand the core calculus of meaning in detail in order to properly write new rules.

This means – in turn – that the topics one utilizes to describe the network must be sufficiently understood by the describer to avoid confusion.

11.2 Lack of contradictions

One specific limitation – due to the need to scale to large data sizes – is that contradictions cannot be expressed in the logical system. There is no provision for any equivalent to the statement that "X is not like Y". One can assert similarity, but not difference. Since the associations are purely syntactic, there is no reason – within the system – that data cannot become contradictory by, e.g., asserting two mutually exclusive relationships for an entity.

11.3 Opportunities for further work

Several key questions remain:

1. Is inference distance the best metric of how related two entities are? Are there other better metrics? Is there a concept of relationship that could aid in measuring distance.

2. How should we handle ternary and n-ary relationships?

3. How can we automatically translate common CMDBs (other than Cfknowledge) into a useful form?

4. How can we relate this work to probabilistic methods for discovering connections?

The search will continue for answers to these questions.

12 Lessons learned

Perhaps the most important lesson learned in this work is that naive approaches to making connections between components do not work properly. This paper describes the 14th prototype. Prototypes 1-13 suffered from a variety of serious problems.

First, tracing connections without considering their meaning gives many *false positives* where the discovered connection is not useful or relevant. For example, one might naively infer from

```
ubuntu | has part | kernel
ubuntu | has part | contributed software
```

that somehow the kernel is related to the contributed software, but that is not particularly useful in troubleshooting.

Second, anything short of real computer logic results in *false negatives*. We tried, e.g., to build connections from known connected components to new ones, breadth-first. This resulted in lost relationships, because some causal relationships are inferred from non-causal ones. For example, consider

```
client workstation | contains | compiler
compiler | has instance | gcc compiler
gcc compiler | requires | linker
```

Because containment is not guaranteed to be causal, starting a walk at '`client workstation`' and looking for causal relationships will never get to '`linker`', even though the inference is that

```
client workstation | can require | linker
```

just because of the choice of starting point for the walk and the fact that there are two non-causal links in the sequence. If we start at '`linker`' instead, then the link will be made, but then other connections may be lost. We were unable to "simplify" the logic without losing connections in this manner.

Third, it is extremely important to keep that logic as simple as possible, so that a human can understand it. The simplest representation seems to be a linear chain of components, with their low-level relationships, where the logic is *not* represented in the chain. In the uses we have developed so far, it is the connections themselves – and not the logic by which they are proven to be connected – that is useful to the end-user.

Fourth, the least specific and most abstract forms of causation are the most useful to reason about. The reason for this is somewhat subtle. In the prototype, one specifies a "pivot" relationship, e.g., '`determines`' or '`can determine`' or '`might determine`', and requests the identities of all components having that relationship to the components that exhibit symptoms. This reasoning works best when that pivot is least specific (e.g., '`might determine`'), because our prototypical architecture specification is always incomplete (just like real architectural specifications).

Our prototype and strategy are not "the solution" to troubleshooting, but rather, utilizes a part of available information that was previously ignored. It is not a replacement for discovering causal links or remembering relationships, but makes relationships more difficult to *forget*.

It is our hope that this demonstration of the utility of this kind of information will encourage people to collect more of it, and in turn encourage all system administrators to utilize configuration management systems (either Cfengine or any other) to define configuration in terms of similar high-level architectural models. The ability of system administrators to think in terms of architectural models – and not this work in particular – is what will actually advance the state of the art.

13 Availability

The prototype is freely available from `http://www.cs.tufts.edu/~couch/topics` . We encourage you to try it with your configuration data and share your experiences with us. Your feedback is important and will help to shape the next generation of these tools and approaches.

14 Author Biographies

Alva Couch is an Associate Professor of Computer Science at Tufts University. He is an author of numerous papers on the theory and practice of system administration, and currently serves as Secretary to the USENIX Board of Directors and chair of the LISA steering committee. He can be reached by electronic mail as couch@cs.tufts.edu.

Mark Burgess is a Professor of Network and System Administration at Oslo University College, Norway. He is the author of Cfengine and several books and papers on system administration, as well as chief technical officer of Cfengine AS. He can be reached by electronic mail as Mark.Burgess@iu.hio.no.

15 Acknowledgments

We would like to thank Oslo University College for generously funding Prof. Couch's extended residence at the University, during which time this work was

done. Shepherd Carolyn Rowland provided invaluable feedback.

References

[1] Bergstra, J., and Burgess, M. A static theory of promises. Tech. rep., arXiv:0810.3294v1, 2008.

[2] Burgess, M. A site configuration engine. *Computing systems (MIT Press: Cambridge MA) 8* (1995), 309.

[3] Burgess, M. On the theory of system administration. *Science of Computer Programming 49* (2003), 1.

[4] Burgess, M. An approach to understanding policy based on autonomy and voluntary cooperation. In *IFIP/IEEE 16th international workshop on distributed systems operations and management (DSOM), in LNCS 3775* (2005), pp. 97–108.

[5] Burgess, M. Knowledge management and promises. *Lecture Notes on Computer Science 5637* (2009), 95–107.

[6] Burgess, M., and Couch, A. Autonomic computing approximated by fixed point promises. *Proceedings of the 1st IEEE International Workshop on Modelling Autonomic Communications Environments (MACE); Multicon verlag 2006. ISBN 3-930736-05-5* (2006), 197–222.

[7] Couch, A., and Burgess, M. Human-understandable inference of causal relationships. In *Proceedings of the First International Workshop on Knowledge Management for Future Services and Networks (KMFSAN10)* (2010), Springer.

[8] Couch, A., and Daniels, N. The maelstrom: Network service debugging via "ineffective procedures". *Proceedings of the Fifteenth Systems Administration Conference (LISA XV) (USENIX Association: Berkeley, CA)* (2001), 63.

[9] Couch, A., and Gilfix, M. It's elementary, dear watson: Applying logic programming to convergent system management processes. *Proceedings of the Thirteenth Systems Administration Conference (LISA XIII) (USENIX Association: Berkeley, CA)* (1999), 123.

[10] Couch, A., Hart, J., Idhaw, E., and Kallas, D. Seeking closure in an open world: A behavioural agent approach to configuration management. *Proceedings of the Seventeenth Systems Administration Conference (LISA XVII) (USENIX Association: Berkeley, CA)* (2003), 129.

[11] Couch, A., and Sun, Y. On the algebraic structure of convergence. *LNCS, Proc. 14th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, Heidelberg, Germany* (2003), 28–40.

[12] Couch, A., Wu, N., and Susanto, H. Towards a cost model for system administration. *Proceedings of the Nineteenth Systems Administration Conference (LISA XIX) (USENIX Association: Berkeley, CA)* (2005), 125–141.

[13] Eiter, T., and Gottlob, G. The complexity of logic-based abduction. *J. ACM 42*, 1 (1995), 3–42.

[14] Forum, T. Information framework (sid). website.

[15] Jensen, F. V., Kjæ rulff, U., Kristiansen, B., Langseth, H., Skaanning, C., Vomlel, J., and Vomlelová, M. The sacso methodology for troubleshooting complex systems. *Artif. Intell. Eng. Des. Anal. Manuf. 15*, 4 (2001), 321–333.

[16] Liberatore, P., and Schaerf, M. Compilability of propositional abduction. *ACM Trans. Comput. Logic 8*, 1 (2007), 2.

[17] Mickens, J., Szummer, M., and Narayanan, D. Snitch: interactive decision trees for troubleshooting misconfigurations. In *SYSML'07: Proceedings of the 2nd USENIX workshop on Tackling computer systems problems with machine learning techniques* (Berkeley, CA, USA, 2007), USENIX Association, pp. 1–6.

[18] Nelson, K. M., Nadkarni, S., Narayanan, V. K., and Ghods, M. Understanding software operations support expertise: a revealed causal mapping approach. *MIS Q. 24*, 3 (2000), 475–507.

[19] Nelson, K. M., Nelson, H. J., and Armstrong, D. Revealed causal mapping as an evocative method for information systems research. In *HICSS '00: Proceedings of the 33rd Hawaii International Conference on System Sciences-Volume 7* (Washington, DC, USA, 2000), IEEE Computer Society, p. 7046.

[20] Nordh, G., and Zanuttini, B. What makes propositional abduction tractable. *Artif. Intell. 172*, 10 (2008), 1245–1284.

[21] Parsons, J. An Information Model Based on Classification Theory. *MANAGEMENT SCIENCE 42*, 10 (1996), 1437–1453.

[22] Parsons, J., and Wand, Y. Emancipating instances from the tyranny of classes in information modeling. *ACM Trans. Database Syst. 25*, 2 (2000), 228–268.

[23] Pepper, S. The tao of topic maps. In *Proceedings of XML Europe Conference* (2000).

[24] Pepper, S. *Encyclopedia of Library and Information Sciences.* CRC Press, ISBN 9780849397127, 2009, ch. Topic Maps.

[25] Steinder, M., and Sethi, A. A survey of fault localization techniques in computer networks. *Science of Computer Programming 53* (2003), 165.

[26] Wang, Y.-M., Verbowski, C., Dunagan, J., Chen, Y., Wang, H. J., Yuan, C., and Zhang, Z. Strider: A black-box, state-based approach to change and configuration management and support. In *LISA '03: Proceedings of the 17th USENIX conference on System administration* (Berkeley, CA, USA, 2003), USENIX Association, pp. 159–172.