# Modern Network Troubleshooting with Declarative Debugging

Anduo Wang
*Temple University*
anduo.wang@gmail.com

Matthew Caesar
*University of Illinois Urbana-Champaign*
caesar@illinois.edu

## 1 Introduction

The broader initiative to modernize computer networks — whether in public Internet infrastructure, hyperscale private WANs and data centers, or the Internet of Things and smart spaces — by transitioning from guesswork-based protocols to provably correct software, primarily through software-defined and programmable networks, has transformed network management. The centralized control software and programmable switches have enabled far greater control — such as fine-grained resource orchestration and accurate traffic engineering. On the other hand, the complexity of the resulting software system has started to resemble the complexity of the protocols we've been trying to shed. When anomalies occur, operators often find debugging difficult, especially if they were not involved in writing the software in the first place. If the complexity of modern networks does not magically disappear, can we at least make them more manageable? Can we make network management truly *declarative* so that, instead of relying on operational network configurations, users can focus on high-level intentions rather than the low-level implementation of protocols and devices?

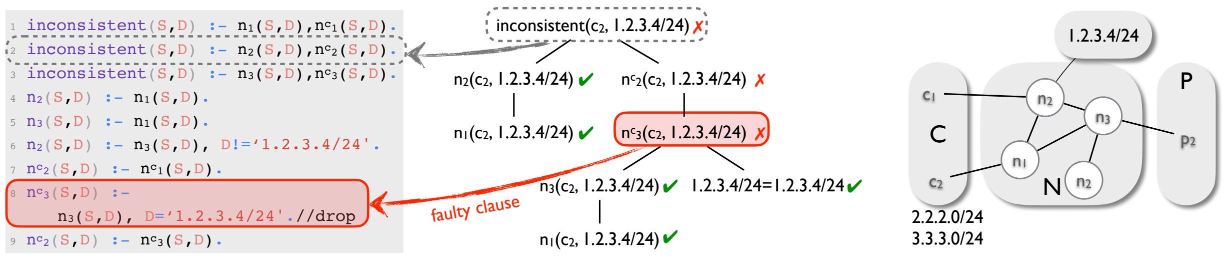
In the era of more capable SDN and programmable networks, achieving this requires more powerful management tools. One important class of tool support that has become both more important and more natural is network troubleshooting with formal methods. Perhaps one of the greatest achievements in this area is the collection of production-grade network verification tools, which outperform manual troubleshooting by orders of magnitude. Nevertheless, stepping outside the tools' comfort zone — verification tasks that are readily recognizable — causes troubleshooting to revert to the same old daunting task, especially if the user lacks deep insight into the network's actual operation. This is not surprising: most formal analysis in these tools is a "batch" activity that is highly optimized for predefined problems, similar to how program analysis is used in compilers for optimization. Consider, for example, the lightning-fast checkers used to verify all reachability properties across entire data centers. But in a live network managed by a human operator, no two problems (symptoms) are the same and no two operators are alike, just like when programmers create and maintain large software systems, no two problems or programmers are alike. A programming-environment-like troubleshooting platform that allows versatile, interactive, and user-centered analysis is still missing in networking.

We take a first step towards a network troubleshooting environment by borrowing from algorithmic program debugging (also known as declarative debugging) — a debugging scheme that originated in logic programming and has since seen widespread applications in imperative languages [2, 4]. Given a symptom — an incorrect behavior produced (computed) by the network, such as unintended forwarding — and a modern network specification, such as a formal executable model in the design phase or the control software that actually drives the network after deployment, we use a declarative debugger to accurately locate the root cause: the specific line of code in the network specification responsible for the error. We emphasize the declarative nature of this approach: the heavy-lifting work of tracing complex network execution and isolating all relevant computations is outsourced to the debugger, the user only needs to provide the intended semantics by answering "yes/no" questions. Indeed, the user acts as an oracle — determining whether an intermediary result is intended (correct) or not — as the declarative debugger navigates through the buggy execution. In the rest of this poster, we use a concrete example to show the feasibility and potential of using declarative debugging as a means to advance network troubleshooting.

## 2 A network diagnosis problem

Figure 1 (c) depicts a network originally introduced by Batfish [3] to motivate multipath consistency verification. We consider a simplified version of diagnosing the center network N for forwarding to N's internal subnet 1.2.3.4/24: the intended forwarding is that the subnet can be reached by the customer network (C), but not the provider (P). To block traffic from P, $n_3$ is configured to drop the packets to 1.2.3.4/24. To enable connectivity with C, $n_1$ and $n_2$ are configured with flow entries that forward traffic from C (e.g., $c_2$) to 1.2.3.4/24. The problem is that $n_1$ is configured with multipath routing by default: as $n_1$ sends packets from $c_2$ to 1.2.3.4/24 through both neighbors $n_2$ and $n_3$, $c_2$ will experience intermittent connectivity.

In addition to multipath consistency verification, which is capable of catching the above error, it also allows for a limited form of root cause diagnosis. Batfish builds an intermediate representation of the network based on Datalog. This representation forms a knowledge base (ontology) of network facts and relations, which can be



```
1 inconsistent(S,D) :- n1(S,D),nc1(S,D).
2 inconsistent(S,D) :- n2(S,D),nc2(S,D).
3 inconsistent(S,D) :- n3(S,D),nc3(S,D).
4 n2(S,D) :- n1(S,D).
5 n3(S,D) :- n1(S,D).
6 n2(S,D) :- n3(S,D), D!='1.2.3.4/24'.
7 nc2(S,D) :- nc1(S,D).
8 nc3(S,D) :-
      n3(S,D), D='1.2.3.4/24'.//drop
9 nc2(S,D) :- nc3(S,D).
```

(a) (b) (c)

Figure 1: Algorithmic program debugging: (a) an NoD-like forwarding program, (b) in the proof tree for the symptom $\texttt{inconsistent}(\texttt{c}_2, '1.2.3.4/24')$: the true nodes (predicates) are marked with ✓, the false ones are labeled with ✗; the false node $\texttt{n}_3^{\texttt{c}}(\texttt{c2}, \texttt{1.2.3.4/24})$ with only true children pinpoints the faulty clause, (c) An example network.

interactively queried to explain a property violation. The Batfish diagnosis is inadequate: a formal definition of "root causes" was never given. The burden of diagnosis — asking the right queries — falls entirely on the user, relying on the user's insight into the network's operational semantics (the correct interpretation of the intermediary NoD forwarding program). It is worth noting that even this limited form of diagnosis was lost after the Batfish implementation moved away from Datalog to Java, where performance was prioritized at the expense of diagnosis [1].

## 3 A solution with declarative debugging

A declarative program diagnoser troubleshoots a symptom (e.g., an incorrect answer or a missing one) by interacting with an oracle — often the programmer — through simple questions, such as whether a specific result (e.g., an immediate computation) was intended or not. The diagnoser builds a representation of the buggy execution and guides the oracle — by asking necessary questions — to systematically explore it and locate the root cause. The main advantage is that, to identify the culprit of a symptom in the code (e.g., a faulty statement), the oracle only needs to provide the what — the program's intended meaning — without knowing the how — the program's actual execution. The clean semantics of logic programming languages, such as Prolog and datalog, makes it particularly amenable to declarative debugging: a clause — a rule $\texttt{h} : -\texttt{b}_1, \cdots, \texttt{b}_n$ that derives its head predicate ($\texttt{h}$) from the predicates in its body ($\texttt{b}_i$'s) — is faulty if it derives (computes) an incorrect result (in the head) from correct ones; In other words, a clause is faulty if it is responsible for introducing the first error into the computation. A symptom is a fact computed by the program that was not intended — that is, it is an incorrect answer. Given a symptom $\texttt{s}$ — that is, an incorrect answer — the declarative debugger only needs to search the execution trace — such as a proof tree — of $\texttt{s}$ to locate the faulty clause.

As an example, consider again Figure 1 (c). A datalog-like specification of network $\texttt{N}$'s forwarding is shown in Figure 1, along with the clauses needed to define the multipath-consistency property. Lines 4-6 define forwarding similar to Batfish (NoD): $\texttt{n}_1(\texttt{S},\texttt{D})$ ($\texttt{n}_2, \texttt{n}_3$ respectively) is true if packets with source $\texttt{S}$ and destination $\texttt{D}$ is at $\texttt{n}_1$; Lines 7-9 defines the impact of dropping a packet: $\texttt{n}_1^{\texttt{C}}(\texttt{S},\texttt{D})$ means the packet is either dropped at $\texttt{n}_1$ or was dropped along a path to $\texttt{n}_1$. With the type of predicates $\texttt{n}, \texttt{n}^{\texttt{C}}$, lines 1-3 defines inconsistency for a pair of source ($\texttt{S}$) and destination ($\texttt{D}$) to be true, if packets can reach and be dropped at any of the nodes ($\texttt{n}_2, \texttt{n}_3, \texttt{n}_3$) simultaneously. For the symptom that traffic from $\texttt{c}_2$ to $'\texttt{1.2.3.4/24}'$ experiences intermittent drop, encoded by $\texttt{Inconsistent}(\texttt{c}_2, '\texttt{1.2.3.4/24}')$, its proof tree is shown in Figure 1 (b). The root of the proof tree is the symptom query, and every intermediary node $\texttt{i}$ is an intermediary result in the execution: $\texttt{i}$ is derived from its children by some clause $\texttt{c}$ in the program. If $\texttt{i}$ is incorrect, the oracle (programmer) only needs to confirm (answer) whether the children (used to derive $\texttt{i}$) are true or false. If all the children are true, we find the culprit $\texttt{c}$ that introduces the first error. To locate the faulty clause, we traverse this proof tree in post-order and find that the first false node with all true children is $\texttt{n}_3^{\texttt{c}}(\texttt{c2}, \texttt{1.2.3.4/24})$. $\texttt{n}_3^{\texttt{c}}(\texttt{c2}, \texttt{1.2.3.4/24})$ is derived by the clause in line 8, which drops traffic to $'\texttt{1.2.3.4/24}'$ at $\texttt{n}_3$ and is therefore the faulty clause we are looking for.

## References

[1] M. Brown, A. Fogel, D. Halperin, V. Heorhiadi, R. Mahajan, and T. Millstein. Lessons from the evolution of the batfish configuration analysis tool. 2023.

[2] R. Caballero, A. Riesco, and J. Silva. A survey of algorithmic debugging. *ACM Comput. Surv.*, 50(4), Aug. 2017.

[3] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein. A general approach to network configuration analysis. NSDI'15, USA, 2015. USENIX Association.

[4] E. Y. Shapiro. *Algorithmic Program DeBugging*. MIT Press, Cambridge, MA, USA, 1983.