# Automated Source Code Extension for Debugging of OpenFlow based Networks

Stefan Hommes, Frank Hermann, Radu State, Thomas Engel
University of Luxembourg, SnT
6, rue R. Coudenhove-Kalergi, L-1359 Luxembourg
Email: {stefan.hommes, frank.hermann, radu.state, thomas.engel}@uni.lu

***Abstract*—Software-Defined Networks using OpenFlow have to provide a reliable way to to detect network faults in operational environments. Since the functionality of such networks is mainly based on the installed software, tools are required in order to determine software bugs. Moreover, network debugging might be necessary in order to detect faults that occurred on the network devices. To determine such activities, existing controller programs must be extended with the relevant functionality. In this paper we propose a framework that can modify controller programs transparently by using graph transformation, making possible online fault management through logging of network parameters in a NoSQL database. Latter acts as a storage system for flow entries and respective parameters, that can be leveraged to detect network anomalies or to perform forensic analysis.**

## I. Introduction

The emerging concept of Software-Defined Networking (SDN) has attracted a lot of attention following the development of OpenFlow [1], a protocol that is used to communicate between network devices and a centralized network controller. Due to the programmability of the network controller, customized code can be executed that adapts to customer specifications. In this paper we propose an approach to instrumenting a variety of controller programs with logging capabilities based on OpenFlow-specific network parameters. This is necessary since networks are vulnerable to different kind of faults, attacks and misconfiguration for which the root cause must be determined promptly in production environments.
Since each program structure is different due to particular application scenarios, source code modification and extension is a tedious and error-prone task. Therefore, the requirements of our approach are to allow:

- Automated source code extension of a wide variety of controller programs to instrument controlling code in order to add logging and debugging facilities **without** modifying the controller itself. This should be possible with user-defined levels of granularity;
- Storing network parameters in a database to allow user-defined tracing, attack and fault detection capabilities; for instance, digital forensics might require complete tracing support, while other scenarios might be more lightweight;
- Transparent storage for the underlying monitoring plane. Since several solutions exist for storing large datasets, we need an approach that can be easily adapted to a particular storage solution. For instance, changing the storage solution from Redis[1] to HBase[2] should be automated and transparent for the end user.

To satisfy these requirements, our paper proposes a framework that leverages *graph transformation* [2] for automated source code extension in combination with an efficient NoSQL[3] database. The source code is extended with additional functionality for logging network parameters, where the user can choose between different levels of logging detail. In our example, we provide a light mode to store a basic set of flow parameters and a forensic mode to additionally retrieve flow statistics.
The paper is structured as follows: In section II, we describe related work in verification and testing of OpenFlow based SDN. Section III describes our concept of logging flow records in a NoSQL database. A short introduction to graph transformation is given in section IV. In section V, we discuss implementation details of the framework. We conclude our paper in section VI.

## II. Related Work

From a historic perspective, the network management community had already addressed Software Defined Networks (SDNs) ten years ago. Research efforts centred around programmable networks and active networking [3], [4], [5] and [6], [7] have prefigured current SDNs by proposing a more heavyweight, though more flexible architecture. We describe in the following some work that concerns debugging of network controller software for OpenFlow based SDN. Similar to a debugger in software engineering, the network debugger ndb [8] was proposed in order to help the operator to determine the root-cause of software faults. Each switch sends so called *postcards*, which are triggered whenever a packets arrives at a switch and contain information about the matched flow entry. This allows the programmer to find errors not only in the controller software itself, but allows also to incorporate the network devices and data plane. Network debuggers are necessary in order to detect errors when software faults were not detected in the first glance. For the programmer, OpenFlow applications need to be analysed before operational deployment in order to detect software faults. The NICE tool

[1] http://redis.io/
[2] http://hbase.apache.org/
[3] http://nosql-database.org/

[9] aims at testing controller programs to discover violations of correctness properties due to bugs in the controller software. A more focussed view on data plane verification is considered in VeriFlow [10]. This approach adds an additional layer between network controller and network devices that checks new rules in real-time before they are deployed in the network. The tool Anteater [11] focusses also on data plane analysis. It represents the collected network topology and forwarding information bases (FIBs) from the network devices as boolean functions. In combination with user-defined network invariants (e.g. loop-free forwarding), a boolean satisfiability problem (SAT) solver performs the analysis. We have done previous work in graph based analysis of communication patterns in [12], [13], [14] without however to address the dynamic aspect and possible transformations as in the current work.

## III. Logging of flow entries

In the following we describe our approach towards a solution for fault detection based on flow entries. The latter are installed to the forwarding table of a switch through an `OFPT_FLOW_MOD` message in case that an incoming packet can not be matched to an existing flow entry. Such match fields can be wildcarded or contain a specific value (e.g. IP address). After a flow entry is expired, statistical information (e.g. received packets/bytes, flow alive time) can be sent to the controller if a respective flag is set. Since every successful connection between two hosts results in a flow and in a cascade of flow entries, such flow traces provide important information for several applications. For instance, flow entries can be leveraged to visualize traffic flows or to realise load balancing by calculating a baseline for the network. Another application where flow traces can be a valuable source of information is code debugging. Due to the flexibility that is given by the controller acting as a network operating system, the functionality of software running on the controller can become increasingly complex and may involve many different program parts. Finding the root cause of an error in the case of an incident can be challenging for a network administrator. Therefore, we propose fields that extend each flow entry with details about the origin of the installation program:

- Unique flow identifier
- Switch identifier
- Path of associated program
- Line in Code

By storing all flow entries with the proposed parameters as *flow records* in a central database, polling traffic to the switches is reduced, while bringing the benefit of having a history of all monitored flows for forensic analysis. It allows also to detect logical errors in the controller code by defining safety invariants which are compared with the relevant flow parameters. Since each flow record in the database contains the program name and line in code, it is possible to determine the program on the controller that installed the flow entry. Furthermore, suspicious flow entries on a switch can be traced back to the program that sent the installation message by their unique identifier for root cause analysis.

### A. Database management

Saving flow entries as flow records in a central database is challenging since huge numbers of them must be inserted in a very short time. A benchmark for the POX controller showed that slightly over 30K new flows per second can be processed [15]. In order to achieve this high scalability and performance, we propose the use of a NoSQL database system having a key-value store. For implementation, we chose the open source database Redis that uses hash types to map between string fields and string values and is used to represent structured objects:

```
[key]:{field -> value, field -> value}
```

Each flow record is defined as a hash object that contains the respective parameters (e.g. IP address, line in code) as field values. A second hash object is generated for each switch and stores all associated flow records by their IDs as field values. The current status of each flow record is specified by a flag that denotes whether it is active (1) or removed (0). An active state defines a flow entry that is currently in use, whereas a removed flow entry has expired or was uninstalled by the controller. The latter can be determined by setting a flag in the `OFPT_FLOW_MOD` message to acknowledge each flow removal. In addition, two fields are reserved as counters that contain the total number of installed and removed flow entries for the respective switch. The described object structures allow for a convenient way of obtaining information about the network with only a small number of database requests.

In order to relieve the burden of the programmer in adding the code for the described logging solution to existing controller programs, we propose dynamically generating code using graph transformation, as displayed in Figure 1. The steps involved are (1) parsing of the source code to derive its abstract syntax tree representation, AST1; (2) executing the graph transformation system on AST1, yielding a resulting abstract syntax tree AST2, where intermediate graphs contain additional temporal structures; and (3) serialising AST2 to derive the extended source code.

## IV. Graph Transformation

The formal technique of algebraic graph transformation [16], [17] generalizes term rewriting techniques used for the transformation of abstract syntax tree structures in formal language theory to the more general case of abstract syntax graph structures. In the present scenario of automated source code extension, this enriched flexibility and the available formal results provide us with additional capabilities for achieving a high level of maintainability and reliability.

In the algebraic approach to graph transformation [17], a graph $G = (V, E, (s, t\colon E \rightarrow V))$ consists of a set $V$ of nodes (vertices), a set $E$ of edges, and the functions $s, t\colon E \rightarrow V$ mapping each edges $e \in E$ to its corresponding source and target nodes, respectively. The approach provides

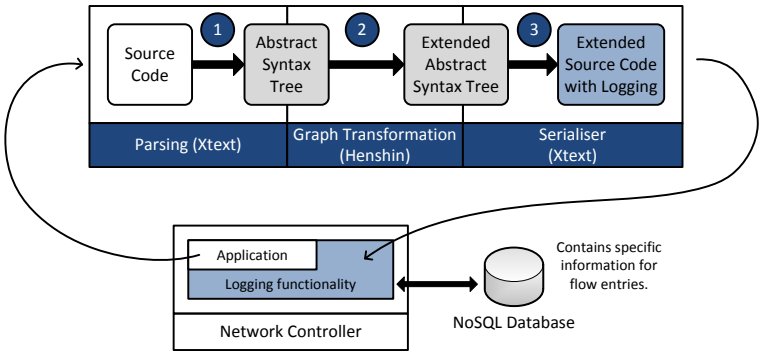


Fig. 1. Automated source code extension by the use of graph transformation for adding logging functionality to controller programs.

formal concepts for attribution and typing including node type inheritance.

A graph transformation step $G \overset{p,m}{\Longrightarrow} H$ (as depicted right) rewrites graph $G$ to $H$ via rule (production) $p$ and match $m$. Such a rule $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ consists of a left hand side (LHS) graph $L$, an interface graph $K$, a right hand side (RHS) graph $R$, and inclusions $l\colon K \hookrightarrow L$, $r\colon K \hookrightarrow R$ (called rule morphisms) relating the rule components. It specifies how a graph structure $L$ found in a graph $G$ is rewritten into $R$, while the context structures around $L$ in $G$ are preserved. Given a match $m : L \to G$ (injective graph morphism) of the left hand side $L$ of rule $p$ into graph $G$ such that $p$ is applicable, the resulting graph $H$ is intuitively derived by removing the parts that are in $L$ but not in $K$ and by adding those that are in $R$ but not in $K$. More formally, the transformation step $G \overset{p,m}{\Longrightarrow} H$ is defined by two pushout diagrams (1) and (2), which means that $G$ is the gluing of $L$ and $D$ via the common part (intersection) $K$ and $H$ is the gluing of $D$ and $R$ via $K$.

$$\begin{array}{ccccc} L & \overset{l}{\longleftarrow} & K & \overset{r}{\longrightarrow} & R \\ {\scriptstyle m}\downarrow & (1) & \downarrow & (2) & \downarrow \\ G & \longleftarrow & D & \longrightarrow & H \end{array}$$

## V. Implementation

Automated source code extension was developed using the Eclipse plugins Xtext [18] and Henshin [19], which are both based on the Eclipse Modelling Framework (EMF)[4]. Henshin is a graph transformation environment. We used its graphical user interface for the visual specification of the graph transformation rules and its graph transformation engine for their execution. The parser and serialiser for Python source code were generated using Xtext with the official EBNF grammar for Python 3.2.

The network controller, POX, and the Redis database ran on a Intel i5 PC (2.50 GHz x 4 cores) with 8 GB of RAM and an Ubuntu 12.04 64-bit OS. The Redis benchmark utility showed a rate of approximately 150,000 requests per second for the `SET` and `GET` database requests. These values can be seen as sufficient, since a single POX controller can handle slightly over 30K flows per second.

We demonstrated a typical application scenario for graph transformation based on the Python controller program `l2_learning.py`[5]. It enables the controller to build a table that maps the source hardware address to the switch port from which the frame was received. If the destination hardware address is unknown, the switch floods the frame out to all ports. If the port of the destination hardware address is determined by the reply, a flow is installed on the switch in order to forward subsequent frames without involving the network controller. In order to extend this program with the described logging capabilities of section III for debugging purposes, we developed two rule sets that can be applied to various controller programs. The user can choose between a *light* and a *forensic* mode. In the first, a user-defined selection of flow entry parameters is saved in the database, which is useful for archiving or simplistic analysis. In the forensic mode, the functionality of the first mode is extended by setting up the switch to deliver statistical information after each flow removal. Such information includes the flow duration and number of received packets, together with the reason for removal (e.g. expired, uninstalled).

Technically, the two rule sets are stored as two transformation units that share a subset of rules. This reuse improves the maintainability of the implementation should additional logging features be incorporated. The source code for the light and forensic mode is available online[6], as well as an adapted screen shot from the graph transformation tool Henshin [19] that displays the graph transformation rule extend_ofp_flow_full as an example. Henshin uses a compact visualisation of rules, where the interface graph $K$ is omitted. The mappings are specified by numbers $[n]$ for the nodes and additionally, by common colouring differing from grey. Thus, the interface graph $K$ is implicitly given by the intersection of the LHS and RHS, i.e., by all numbered elements. In the screen shot, the RHS $R$ is shown with a decreased scaling factor. It contains 190 nodes and has a height of 18. The text boxes in the bottom part of the figure contain source code fragments

[4]http://www.eclipse.org/emf/

[5]https://github.com/noxrepo/pox/tree/betta/pox/forwarding/l2_learning.py

[6]https://github.com/shommes/POX.git

that correspond to the upper four framed components of $R$. Note that only the elements in the left hand side $L$, its counterparts in $R$ (zoomed component) and its adjacent edges were specified manually, while the remaining graphical part was inserted using the automated import mechanism of the Henshin tool. The complete set of transformation rules for extending Python code with logging information contains 13 rules and was created by taking relevant example source code fragments, importing them into the Henshin GUI and adapting those parts that are specific to the example used in order to derive general purpose rules.

Based on the available formal results and tool support, we were able to show certain correctness properties [20]. First of all, the formal concept of typed attributed graph transformation ensures that at each step, the intermediate graphs are correctly typed and satisfy the structural conditions of the type graph [17]. Moreover, we formally proved that the transformation system for source code extension always terminates and yields unique results. To do this, we used the tool support in AGG [21] based on static termination and confluence criteria [17] with some slight but direct extensions based on the concept of layering and the tree structure of the input abstract syntax tree. Based on the consistency conditions in [22], we proved that the resulting output always has a tree structure.

## VI. Conclusion and future work

This paper describes an approach to instrument network controllers in OpenFlow with logging capabilities that allow network debugging. In order to avoid manual code modification by the programmer, we propose using graph transformation, which allows automated source code extension based on user-defined transformation rules. This concept, which can easily be modified for additional use-cases in similar scenarios, was further demonstrated to instrument controller software in order to log flow entries as flow records in a database system. Latter serves as a storage solution, which allows both forensic analysis and anomaly detection based on the archived flow records. We will determine in future work the potential of the logged information for fault management and in order to detect network attacks.

## Acknowledgement

Supported by the Fonds National de la Recherche, Luxembourg (PhD-09-188).

## References

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008.

[2] H. Ehrig, K. Ehrig, A. Habel, and K.-H. Pennemann, "Theory of Constraints and Application Conditions: From Graphs to High-Level Structures," *Fundamenta Informaticae*, vol. 74, no. 1, pp. 135–166, 2006. [Online]. Available: http://fi.mimuw.edu.pl/vol74.html

[3] R. Stadler and B. Stiller, Eds., *Active Technologies for Network and Service Management, 10th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, DSOM '99, Zurich, Switzerland, October 11-13, 1999, Proceedings*, ser. Lecture Notes in Computer Science, vol. 1700. Springer, 1999.

[4] J. Biswas, A. A. Lazar, J. F. Huard, K. Lim, S. Mahjoub, L. F. Pau, M. Suzuki, S. Torstensson, W. Wang, and S. Weinstein, "The IEEE P1520 standards initiative for programmable network interfaces," *Comm. Mag.*, vol. 36, no. 10, pp. 64–70, Oct. 1998.

[5] M. Brunner, B. Plattner, and R. Stadler, "Service creation and management in active telecom networks," *Commun. ACM*, vol. 44, no. 4, pp. 55–61, Apr. 2001.

[6] J. Smith and S. Nettles, "Active networking: one view of the past, present, and future," *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, vol. 34, no. 1, pp. 4 –18, feb. 2004.

[7] D. Tennenhouse, J. Smith, W. Sincoskie, D. Wetherall, and G. Minden, "A survey of active network research," *Communications Magazine, IEEE*, vol. 35, no. 1, pp. 80 –86, jan. 1997.

[8] N. Handigol, B. Heller, V. Jeyakumar, D. Maziéres, and N. McKeown, "Where is the debugger for my software-defined network?" in *Proceedings of the first workshop on Hot topics in software defined networks*, ser. HotSDN '12. New York, NY, USA: ACM, 2012, pp. 55–60.

[9] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford, "A NICE way to test openflow applications," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, ser. NSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 10–10.

[10] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey, "Veriflow: verifying network-wide invariants in real time," in *Proceedings of the first workshop on Hot topics in software defined networks*, ser. HotSDN '12. New York, NY, USA: ACM, 2012, pp. 49–54.

[11] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King, "Debugging the data plane with anteater," in *Proceedings of the ACM SIGCOMM 2011 conference*, ser. SIGCOMM '11. New York, NY, USA: ACM, 2011, pp. 290–301.

[12] J. François, S. Wang, R. State, and T. Engel, "Bottrack: tracking botnets using netflow and pagerank," in *NETWORKING 2011*. Springer Berlin Heidelberg, 2011, pp. 1–14.

[13] C. Wagner, G. Wagener, R. State, and T. Engel, "Malware analysis with graph kernels and support vector machines," in *Malicious and Unwanted Software (MALWARE), 2009 4th International Conference on*. IEEE, 2009, pp. 63–68.

[14] H. J. Abdelnur, R. State, and O. Festor, "Advanced network fingerprinting," in *Recent Advances in Intrusion Detection*. Springer Berlin Heidelberg, 2008, pp. 372–389.

[15] "About POX," 2012. [Online]. Available: http://www.noxrepo.org/pox/about-pox/

[16] G. Rozenberg, *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.

[17] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer, *Fundamentals of Algebraic Graph Transformation*, ser. EATCS Monographs in Theor. Comp. Science. Springer, 2006.

[18] *Xtext - Language Development Framework – Version 2.3*, Eclipse Consortium, 2012, http://www.eclipse.org/Xtext/.

[19] T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer, "Henshin: Advanced concepts and tools for in-place EMF model transformations," in *Proc. MoDELS'10*, ser. LNCS, vol. 6394. Springer, 2010, pp. 121–135.

[20] F. Hermann, S. Hommes, R. State, and H. Ehrig, "Correctness of source code extension for fault detection in OpenFlow based networks," Technische Universität Berlin,Fakultät IV, Tech. Rep. 2013-xx, 2013, to appear. [Online]. Available: http://user.cs.tu-berlin.de/ frank/Papers/HHSE13.pdf

[21] *AGG*, TFS-Group, TU Berlin, 2012, http://www.tfs.tu-berlin.de/menue/forschung/software/.

[22] E. Biermann, C. Ermel, and G. Taentzer, "Formal foundation of consistent EMF model transformations by algebraic graph transformation," *Software and Systems Modeling (SoSyM)*, vol. 11, pp. 1–24, 2011.