Modular SDN Compiler Design with Intermediate Representation

Hao Li, Chengchen Hu, Peng Zhang, Lei Xie
Xi'an Jiaotong University[*]

CCS Concepts

•**Networks → Programming interfaces;**

Keywords

Software-Defined Networks; Intermediate representation;

1. INTRODUCTION

SDN decouples the control plane that programs network controls with high-level *languages*, and the data plane that exercises packet forwarding with low-level *rules*. After years of development, SDN is gradually evolving into an era that various languages (Frenetic, Maple, Merlin, P4, *etc*) and rules (OpenFlow, PoF, ACI, *etc*) coexist.*SDN compiler* translates programs written by some languages into rules, however, one SDN compiler only bridges one language and one rule specification. This brings the following two major problems.

First, a piece of SDN program written in one specific language cannot interoperate cross multiple data planes. As a result, programs that produce OpenFlow rules cannot be used in POF-based network. Moreover, many optimization techniques, which improve the performance, reduce the overhead, and ensure the correctness, are also rule-specific. For example, VeriFlow [3] can only verify OpenFlow rules.

Second, even for the data plane using a single rule specification, programs written with different languages may lead to unsolvable conflicts. The reason is that we cannot resolve the conflicts by simply merging the programs as they are in different languages, or by analyzing rules as intents of applications are lost during the compilation. For instance, to route the same flow, a policy chaining program written with Merlin and another traffic engineering program written with Pyretic may specify different forwarding paths. Arbitrarily overwriting the conflicting rules may break the original intents, *e.g.*, waypoint traversal, shortest-path routing, *etc*.

The root cause of the above problems is the incomplete decoupling of languages and rules. To end this situation , we introduce *Semantics Rule (SR)*, a rendezvous point for languages and rules, which is inspired by the intermediate representation (IR) for PC compilers. Using SR, SDN programs will be first compiled into SRs (front end), applied with common optimizations (middle end), and finally translated into low-level rules (back end). As a result, programs written with different languages can inter-operate on multiple networks that enforce different rules. In addition, optimization applied at SRs can be targeted to all rule specifications.

Table 1: Syntax of the Semantics Rule (SR).

SR	*sr* ::=	<*scope*, *cons*, *dof*>
Scope	*scope* ::=	*match fields*
Constraint	*cons* ::=	**forward** | **drop** | **rewrite** | **count**
DOF	*dof* ::=	**forward**(*port*) | **towards**(*sw*) | **forbid**(*sw*)

2. SEMANTICS RULES AS THE IR

We define SR as a 3-tuple <*Scope*, *Constraint*, *Degree of Freedom (DOF)*>, which specifies constraints on network resources, and high-level intents that a set of functions can operate on. Table 1 shows the simplified syntax of SR.

Scope specifies the resource that an SR operates on, analog to the *match fields* specified by an OpenFlow rule.

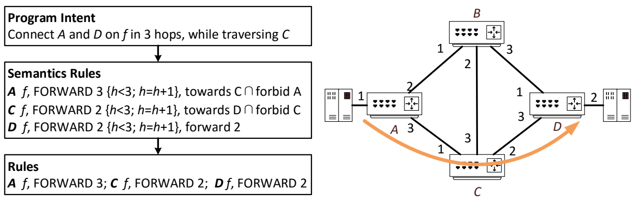
Constraint specifies actions on the Scope, *e.g.*, forwarding to a port, dropping, rewriting a field, or counting. The Constraint is associated with a pair of $(guard, action)$, where $guard$ consists of performance predicates, like #hops$\leq$5, delay$<$10ms, *etc*. The Constraint is valid only if its guard is satisfied, and the action will be triggered when the Constraint is selected. Such pair improves the expressiveness by lifting the semantics level of SR, *i.e.*, it can express the context-sensitive constraints besides the forwarding targets.

DOF specifies the alternative space for the Constraint: an SR remains semantically the same if its Constraint is replaced by another point in the DOF. DOF allows SRs to relay high-level intents specified by operators. For example, `forward(`X`)` defines a singleton DOF, "forward to port X"; `towards(`X`)` means "forward to any port that can finally lead to switch X"; `forbid(`X`)` means "forward to any switch except switch X".

As seen above, SR uses DOF to be more expressive, so that common intents (*e.g.*, forward a packet towards a des-

[*]This paper is supported by the NSFC (No.61272459), Program for New Century Excellent Talents in University(NCET-13-0450), the Fundamental Research Funds for the Central Universities.

SIGCOMM '16, August 22–26, 2016, Florianopolis, Brazil
© 2016 ACM. ISBN 978-1-4503-4193-6/16/08…$15.00
DOI: http://dx.doi.org/10.1145/2934872.2959061



Program Intent
Connect *A* and *D* on *f* in 3 hops, while traversing *C*
Semantics Rules
A *f*, FORWARD 3 {*h*<3; *h*=*h*+1}, towards C ∩ forbid A
C *f*, FORWARD 2 {*h*<3; *h*=*h*+1}, towards D ∩ forbid C
D *f*, FORWARD 2 {*h*<3; *h*=*h*+1}, forward 2
Rules
A *f*, FORWARD 3; ***C*** *f*, FORWARD 2; ***D*** *f*, FORWARD 2
B
A
C
D
1
2
3

Figure 1: Top-down compiling process with SR.

tination host) of programs/applications can be largely preserved. In addition, the DOF keeps simple and general in semantics, so that SR is independent of existing languages. Finally, SR uses a general form of Scope and Constraint to achieve rule neutrality, meaning that SRs can be readily translated into any rule forms, *e.g.*, OpenFlow rules.

In practice, a front end compiles the program into a sequence of SRs. The execution process of SRs is to trigger Constraint from the source to the destination for each Scope. The sequential execution ensures the context-sensitive indicators are correctly updated by the guards and actions. Figure 1 depicts the top-down compiling process of a simple program. The SRs use `towards` to constrain the critical nodes along the path, *i.e.*the destination and the waypoints, and use `forbid` to avoid loops.

3. CASE STUDY

In the following, we show how to leverage SR for conflict elimination in cross-language programs.

Due to the language diversity, SDN controllers commonly detect conflicts at the rule level, *e.g.*, checking whether two rules have overlapped matching fields but conflicting actions. When detected, the controller needs to modify one rule to resolve the conflicts, which may break the intents of programs. As another approach, PGA [4] detects conflicts at the language level, and thus can reconcile programs that conflict with each other while preserving their intents. However, PGA only works for the *same-language* programs (*i.e.*, policy graph). Here, we show how SR can eliminate conflicts, while preserving intents of *cross-language* programs. Consider a program indicates a different path for f in Figure 1: $A \rightarrow B \rightarrow D$, where B is a waypoint. The conflicts can be detected in A, where two SRs have the same Scope but different Constraints. The elimination process is to find the DOF's intersection of the conflicting SRs, *i.e.*, `towards`$(B) \cap$ `towards`(C). If the intersection is not empty, then any point in the intersection would be a solution to the new Constraint of both SRs. Note that forwarding to B in A is in the DOF of `towards`(C), because B can later forward the flow to C. Therefore, the final solution can be $A \rightarrow B \rightarrow C \rightarrow D$. The back end then translates the reconciled SRs into non-conflicting rules.

4. EVALUATION

Program interoperability. We prototype a SR front end by modifying the RYU controller to generate SRs instead of OpenFlow rules, and a simple back end to translate SRs to PoF rules. We run three RYU programs on our front end for all-pair reachability, monitoring and flow counting, respectively. The generated PoF rules work correctly in the network managed by RYU, thereby proving the interoperability of RYU programs and PoF rules.

Table 2: Conflicts elimination results.

	#origin. paths	#origin. conflicts	#reconciled paths	#broken paths	time cost(s)
CE	100	122	93	0	0.194
CV	100	122	55	33	0.101
CE	500	972	421	0	0.949
CV	500	972	229	192	0.701
CE	1000	1589	807	0	1.484
CV	1000	1589	331	488	1.113
CE	2000	1854	1552	0	1.988
CV	2000	1854	587	969	1.309

Conflict elimination. We use a program to randomly connects hosts with random waypoints and #hops constraints in the Stanford backbone [5], which generates 100∼2000 paths with 122∼1854 conflicts respectively. We involve a rule-level composition approach CoVisor [1] (CV) into the evaluation, and use Hassel [2] to check the consistency of the reachability, waypoints and #hops constraints after the elimination. CE and CV are implemented in Python. Table 2 shows the elimination results, where the broken path is the path that breaks the original intents. Notice that some conflicts are not resolvable (*i.e.*, empty intersection of DOFs), and the #broken paths excludes such conflicts. From the result, the rule-level CV only composes conflicting rules by priorities, which is likely to violate the intents, while CE does not break any path that can co-work with others. Also, it takes less than 2s to perform CE with ∼2K conflicts. CV is slightly faster than CE because it only prioritizes the rules locally, but not trace the routing path to preserve the intents.

5. CONCLUSION AND FUTURE WORK

This paper has presented Semantics Rule (SR) as the Intermediate Representation (IR) for SDN compilers. We have explored an interesting case using SR that how it can help resolving conflicts for programs written with different languages, which is infeasible by only analyzing rules.

Our further work includes developing SR's representation, especially the annotations of DOF, to better cover the general case in network behaviors, and implementing more optimization techniques with SR. Furthermore, the northbound of the SDN controllers can be unified to a well-designed IR, instead of a "perfect" SDN language. This leads to a purer SDN control model, that the controller can offload the logic programming to the language interface, *i.e.*, compiler front end, and focus on collecting and managing the concrete information from the operators and switches. Design and implement such modular controller is also our future work.

6. REFERENCES

[1] X. Jin, J. Gossels, J. Rexford, and D. Walker. Covisor: A compositional hypervisor for software-defined networks. In *USENIX NSDI*, 2015.

[2] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *USENIX NSDI*, 2012.

[3] A. Khurshid, W. Zhou, M. Caesar, and P. Godfrey. Veriflow: verifying network-wide invariants in real time. In *USENIX NSDI*, 2013.

[4] C. Prakash, J. Lee, Y. Turner, J.-M. Kang, A. Akella, S. Banerjee, C. Clark, Y. Ma, P. Sharma, and Y. Zhang. PGA: Using graphs to express and automatically reconcile network policies. In *ACM SIGCOMM*, 2015.

[5] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. Automatic test packet generation. In *ACM CoNEXT*, 2012.