View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by University of Szeged

Axiom-based property verification for P4 programs

Gabriella Tóth, Máté Tejfel

Abstract: We produce an axiom-based program properties verification method for P4 programs. P4 is a special, domain specific, declarative programming language to develop network packet forwarding. P4 is quite new, and different from general-purpose programming languages so it is an important idea to understand its behavior and to verify it. The operational semantics of P4 is reachable in $\mathbb{K}$ framework, so there is an opportunity to do its verification based on its operational semantics, but it is a low level solution, therefore the proof of complex properties can be too difficult and costly. So we would like to verify the program in a higher abstraction level, in which we would introduce axioms, which are correct in the operational semantics. Using these axioms we can create easier and more transparent proof.

Keywords: P4, $\mathbb{K}$ framework, verification, operational semantics

Introduction

This paper analyze P4 programs and program properties. We chose P4, because it is a new, special, domain specific, declarative programming language, which is created for developing network packet forwarding. P4 has a reachable operational semantics in $\mathbb{K}$ framework, so we can start the verification based on these semantics rules. But the disadvantage of this way is that, it is a low level solution, therefore the proof of complex properties can be too difficult and costly. Consequently, we would like to create a higher abstraction level with introducing operational semantics based axioms, to make an easier way for proof and to reach proof's end sooner.
The above mentioned $\mathbb{K}$ framework is an executable semantic framework, in which we can add any program language's syntax and operational semantics for verify programs with symbolic execution [1][2]. The $\mathbb{K}$ uses configurations to follow the program states, and we can give the semantics rules as a relation from configurations to configuration. The P4 operational semantics is given in this way, so our axioms will be developed in the same style. Using these axioms we can produce an axiom-based program property verification method for P4 programs.

The P4 language

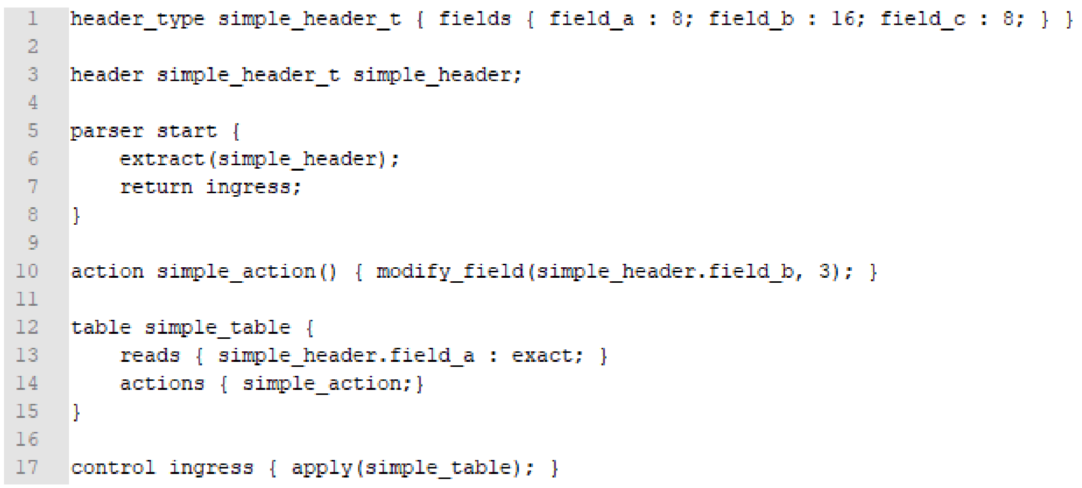
P4 language is a new, domain specific, declarative programming language, which is created for a target independent, protocol independent and reconfigurable way to develop network packet forwarding [3][4]. The board members of the P4 language consortium come from such a famous institutions as Stanford University, Princeton University and Google. So far, there are two version of the language, the $P4_{14}$ and the $P4_{16}$. $P4_{14}$ has a available operational semantics in $\mathbb{K}$ framework [5], so in the paper this version will be used.

An example for P4 program

In Figure 1. we can see a simple P4 program. This example code snippet demonstrated those language elements of P4 that are essential for understanding our results presented in this paper. To give the types of headers is an important part to handle the packets. We can create these elements as a *header_type*, with the name and contained fields and the bit-width of fields. With these header types we can define header instances, with which we can handle the real headers in runtime. In our example these elements appear in the first and the third line. In the first line one header type is defined, which name is *simple_header_t*, and has 3 field, the *field_a* and *filed_c* with 8 bit-width and the *field_b* with 16 bit-width. This type has an instance, with *simple_header* name (line 3).

We need to have parser in our programs, which gives how the first part of the packet have to be divided into headers. In the example the parser (lines 5-8) unwraps the header *simple_header*.

The actions contain the operations which can be executed in the headers. There are predefined primitive actions, and the developer can also compose new actions build up from sequences of primitive actions. In Figure 1. there is only one action named *simple_action*, which calls one *modify_field*. The



```
 1  header_type simple_header_t { fields { field_a : 8; field_b : 16; field_c : 8; } }
 2
 3  header simple_header_t simple_header;
 4
 5  parser start {
 6      extract(simple_header);
 7      return ingress;
 8  }
 9
10  action simple_action() { modify_field(simple_header.field_b, 3); }
11
12  table simple_table {
13      reads { simple_header.field_a : exact; }
14      actions { simple_action;}
15  }
16
17  control ingress { apply(simple_table); }
```

Figure 1: A simple P4 input file

modify_field is a primitive action, and it tries to change the first parameter's value for the second parameter, so here, we try to change the value of the field named *field_b* of the header named *simple_header* to 3.

The declarations of P4 'match+action' tables define what actions can execute based on the processed packet's values. This element has a name, a 'reads' and an 'actions' part. In the 'reads' we can show those header instance's field, which we would like to use during matching, and the type of matching. In the 'actions' part there are the set of executable actions. In our simple program, there is one 'match+action' table (lines 12-15) as *simple_table*, and it would like to match the *simple_header*'s *field_a* field in the exact way, and it can execute one action, the above mentioned *simple_action*.

After the analysis of the packet, the next step is the execution of the control function starting with the ingress control function. These functions define the sequence of the tables' execution for the packet, and define how to continue the execution after a matching. In our simple program there is only one ingress control function, which calls our table's execution in the most simple way.

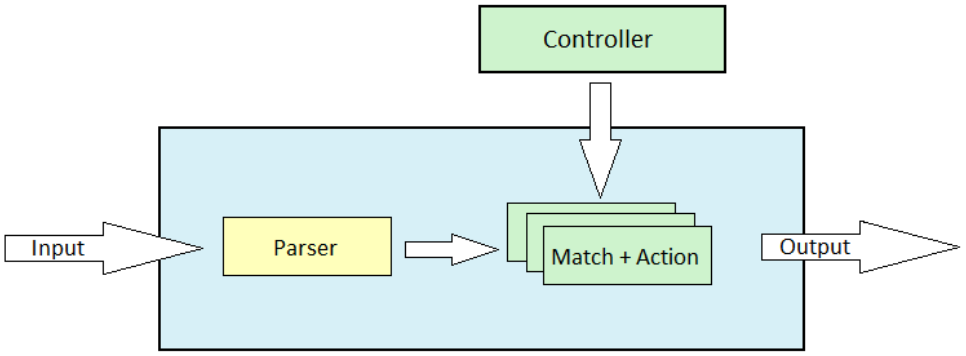


Figure 2: A simplified illustration for P4 behavior

The P4 programs' behavior

P4 has a different behavior as other common programming languages. P4 programs contain only declarations about the packets' headers, parsers, actions, etc. In Figure 2, we can see a simplified illustration for its conduct. First, the packets go through the parser and that creates a set, which contains valid header instances for the packet. Initially 'match+action' tables are empty, just those information is given, which is in the declaration. There is an external controller, which fills these tables with concrete pairs. One pair contains a matching value and the action, which have to be executed when the matching is fine. The uploaded 'match+action' tables operate on parser made set, and this process is based on the control functions. After all processing finished, the resulting packet will be transmitted.

Motivation

A simple way for the verification, if we analyze concrete P4 programs, like a *modify_field* effect in a given value of a header instance's field. For example in Figure 1, there is only one action, which modify the value of *field_b* to 3 (line 10). We could examine the value's changing. If in the initial state *field_b* has any given value, then in the final state its value will be the same or 3. A program property, where we work with concrete values, and concrete elements, can be proved easily using the operational semantics.

In the next step, we can try to go further, and generalize the property. If initially we have a given field with any value and a program contains one *modify_field* action for this field with a new value, then the program would go to a final state, where the value of the given field is the new one. Using operational semantics's rules it would be a complex proof, because it would work with meta-variables over the value and field, with which more calculations of inside steps would not be obvious. There is another properties, which could be examined, if initially we have a given field and a program contains only one *modify_field* action with another name of fields, then the program would go to a final state, where the value of the given field will be the same. It seems a similar statement, but its proof is really hard using only the operational semantics's rules.

In $\mathbb{K}$ framework for the symbolic execution we need to give a file, which contains the behavior of controller, so we need to give the concrete filling of the 'match+action' tables. In the proofs of previous properties, which uses only operational semantics, there would be rules which based on the controller's behavior. The controller can only call actions, which were declared in the used P4 program, therefore an even further way to say statements about a P4 program is that if we do not deal with the controller, and say properties only about P4 declarations. For instance, a statement could say, if we have actions, which contains only *modify_field* primitive actions, and there is a field, which is not appear in any first parameter of *modify_field*s, then the value of this field won't change after any execution, it does not matter how the tables looks like, and what type of actions will be run, because we do not have any action, which would change its value. In this point, we need to go a higher abstraction level, because these type of statements cannot be proved using only operational semantics. In this level instead of symbolic execution, our prove based on some operational semantics's rules and axioms, or in an even higher level only axioms with deduction rules. In this way it can be an axiomatic semantics [6]. The introduced axioms will be defined and proved based on the operational semantics, and sometimes the operational semantics's structure is not enough for the proof, so we will supplement the rules and configurations.

The 'Unchanged Field' axiom

In the example we analyze the property, when the value of examined field is unchanged. Using only the operational semantics's rules, the proof of this property is complicated, so we introduce an axiom for easier proof, which contains conditions in the upper part, and a Hoare triple in the lower part:

$$\frac{f \in \mathit{instanceFields}(S) \text{ and } f \notin \mathit{modifiedFields}(S)}{\{f = X\}\; S\; \{f = X\}}$$

In the axiom, S is the examined program, X indicates any value of a field, $\mathit{instanceFields}(S)$ is a set of fields, which appear in the program and $\mathit{modifiedFields}(S)$ is a set of fields, which value can be changed during the program run. The *modifiedFields* contains those fields, which appear in the first parameter of any primitive actions, because a field's value can be changed only in this way. A field is identified by the instance, which contains it, so an element of the two sets has *instanceName.fieldName* format.

The given axiom expresses that, if there is a field in the program and it is not in the *modifiedFields* set, after the program run, its value will be the same as in the initial state.

Using this axiom in the case of the program introduced by Figure 1 (sign it with S1), we can easily prove the stability of $\mathit{simple\_header.field\_a}$ and $\mathit{simple\_header.field\_c}$. The two sets are computable:

$\mathit{instanceFields}(S1) = \{\mathit{simple\_header.field\_a}, \mathit{simple\_header.field\_b}, \mathit{simple\_header.field\_c}\}$

$modifiedFields(S1) = \{simple\_header.field\_b\}$.

The axiom's conditions are true for $simple\_header.field\_a$ and $simple\_header.field\_c$, so these fields will not change during the execution of $S1$ program.

Acknowledgements

This work has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002).

References

[1] A. Ştefănescu, D. Park, S. Yuwen, Y. Li, G. Roşu: Semantic-Based Program Verifiers for All Languages. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA'16), pages 74-91. ACM, Netherlands, 2016

[2] G. Roşu: $\mathbb{K}$: A Semantic Framework for Programming Languages and Formal Analysis Tools. In *Dependable Software Systems Engineering*, pages 186-206. IOS Press, Netherlands, 2017

[3] The P4 Language Consortium: The P4 Language Specification *https://p4.org/p4-spec/p4-14/v1.0.4/tex/p4.pdf*, 2017

[4] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, D. Walker: P4: Programming Protocol-Independent Packet Processors. In *ACM SIGCOMM Computer Communication Review*, pages 87-95, ACM, USA, 2014

[5] The P4 Language's operational semantics in $\mathbb{K}$ framework: https://github.com/kframework/p4-semantics

[6] R. D. Cameron: Axiomatic Semantics of Programming Languages. 2002