Acta Cybernetica **25** (2021) 537–554.

Component-Based Error Detection of P4 Programs*

Gabriella Tóth[a] and Máté Tejfel[b]

Abstract

P4 is a domain-specific language to develop the packet processing of network devices. These programs can easily hide errors, therefore we give a solution to analyze them and detect predefined errors in them. This paper shows the idea, which works with the P4 code as a set of components and processes them one by one, while calculating their pre- and postconditions. This method does not only detect errors between the components and their connections, but it is capable to reveal errors, which are hidden in the middle of a component. The paper introduces the method and shows its calculation in an example.

Keywords: P4, error detection, component

1 Introduction

We introduce a component-based formal method to detect errors in P4 programs. The antecedent of the method is an error detection, which is based on a rule system [13]. With this idea, we approach the detection from backward and process the code from the smallest units to the biggest ones. This solution can not only check the error possibilities but give additional information about the code for the developer. The prototype of the tool, which is based on the method, is being implemented[1].

P4 programs We work with the P4 language [5, 2, 6, 1], which is a domain-specific programming language to develop the packet processing of network devices. When a packet arrives at a device as a bitstream, the P4 program gets that bitstream as the input and starts to work with it. In Figure 1 there is an example program. P4 programs work with the header information of a network packet. The developers can define what kind of header information they work with (rows 1-16). P4 programs have three main processing parts: parser, modifier, and deparser. The

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

[a]Faculty of Informatics, Eötvös Loránd University, Budapest, Hungary, E-mail: kistoth@inf.elte.hu, ORCID: 0000-0001-9657-7231

[b]Department of Programming Languages and Compilers, Eötvös Loránd University, Budapest, Hungary, E-mail: matej@inf.elte.hu, ORCID: 0000-0001-8982-1398

[1]https://github.com/tothgabi/ELTE_P4_Analyzer

DOI: 10.14232/actacyb.289731

```
1  header ethernet_t {
2      bit<48> dstAddr;
3      bit<48> srcAddr;
4      bit<16> ethernetType;
5  }
6
7  header ipv4_t {
8      bit<8> ttl;
9      bit<32> srcAddr;
10     bit<32> dsttAddr;
11 }
12
13 struct headers {
14     ethernet_t ethernet;
15     ipv4_t ipv4;
16 }
17
18 parser MyParser(...) {
19     state start {
20         transition parse_ethernet;
21     }
22
23     state parser_ethernet {
24         packet.extract(hdr.ethernet);
25         transition accept;
26     }
27 }
```

```
28 control MyIngress(...) {
29     action ipv4_create1 (bit<32> dstAddr) {
30         hdr.ipv4.srcAddr = hdr.ethernet.dstAddr;
31         hdr.ipv4.dstAddr = dstAddr;
32         hdr.ipv4.setValid();
33     }
34
35     action ipv4_create2 (bit<32> dstAddr) {
36         hdr.ipv4.setValid();
37         hdr.ipv4.srcAddr = hdr.ethernet.dstAddr;
38         hdr.ipv4.dstAddr = dstAddr;
39     }
40
41     action drop() { mark_to_drop(standard_metadata); }
42
43     table t {
44         key = { hdr.ethernet.dstAddr : exact; }
45         actions = {ipv4_create1; ipv4_create2; drop; }
46     }
47
48     apply { t.apply(); }
49 }
50
51 control MyDeparser(...) {
52     packet.emit(hdr.ethernet);
53     packet.emit(hdr.ipv4);
54 }
```

Figure 1: Example P4 code

parser (rows 18-27) gets the input and extracts the header information from the packet. The modifier part (rows 28-49) modifies the header information – based on match-action tables, the content of which comes from an external controller – and after the modification, the deparser (rows 51-54) creates the new packet with the calculated header information to forward it to the network.

Software-defined networks have two main parts: the control plane – which controls the network traffic – and the data plane – which describes the process of a network packet. As we can see from the example, P4 programs can define the data plane. Therefore, the behavior of the programs depends on the control plane, which cannot see from the P4 code – for example, there is a program structure, the match-action table, which is only a frame in the code, and its content is filled by an external controller during the execution.

2 Related Work

There are different approaches to analyze P4 programs.

There are tools, which work with the previous version of P4 – the $P4_{14}$ – although the main concept of these projects is similar to the new version. One of them is P4V [9], which deals with errors caused by reading or writing of invalid headers, the arithmetic overflow, and proper accessing of header stacks. It creates an annotated program from the input source and verifies it by checking the correctness of a first-order formula, which describes the execution of the program.

Vera [11] is another one, which works with symbolic execution, which starts with the generation of the parsable input packet, and then checks all possible execution path with the Symnet [12] static analysis tool. They investigate similar error cases,

for example, header validity, implicit drops.

Assert-P4 [8] also works with $P4_{14}$, and uses symbolic execution too, although they check program-specific errors, which can be described by the developer. Therefore it can check errors, which can not be detected only from the simple P4 source, but from an annotated P4 code. It translates the annotated P4 code to a C-model, and then it checks the different properties, which were given in the code, with symbolic execution.

There are other tools, which work with the actual version of P4 – the $P4_{16}$. There is an approach, which does data flow analysis of P4 programs [4]. In this research, they used an extended version of a control-flow graph, the def-use graph. For all possible execution paths, they execute the data flow analysis, which collects the usage of the different variables and header information. Based on this collected data, their solution can detect errors like using an undefined field.

Although the previous projects are based on static analyzes, there are other solutions too. P4RL [10] does runtime verification, therefore it is able to analyze behaviors which are come from the control plane. The behavior of the network should be described by the user, therefore it can check if all of them are correct during the execution.

Another approach is SafeP4 [7], which is a domain-specific language, which was created to develop correct P4 programs.

Our solution works with the new version of P4 – the $P4_{16}$. It is a static analyzer method, which uses only the P4 code as an input. Based on the code, we can extract the specification of the program, as pre- and postconditions, and we detect errors by analyzing the pre- and postconditions of the different components of the program and checking their relationship with the specification of the program. We detect errors, which can be caused by the usage of invalid header and uninitialized fields.

3 Motivation

P4 gives a new possibility for the protocol-independent development of the programmable switches. However developers can create more flexible programs, it is easier to make a mistake in them. That is the reason, why we need a tool, which can detect errors in our code.

As it was mentioned in the Introduction, the P4 programs do not define the whole behavior of the network, but a part of it. Therefore, some verification tools expect annotated programs and additional information about the planned behavior of the program, and they can check if those properties are correct or not. We would like to detect errors based on only the raw P4 code, therefore the developers will not have any other job besides developing the P4 program.

We would like to create a tool, based on our theoretical results, to help the developers to write correct P4 programs. In this first version of the method, we concentrate only on those errors to detect, which are caused by using invalid headers and uninitialized fields. Using them can cause undefined behavior by unknown values. However, we still work with the subset of the P4 language, but we plan to

extend the rule system to work with a bigger language and detect more error cases.

This method is based on the precondition and postcondition of the program units. As we see in Figure 1, there are different programming structures in P4: actions, tables, control functions. It works with them one by one, and calculates condition pairs for each of them, starting from the smallest ones – like actions – finishing with the biggest ones – like control functions.

4 Method of detection

The input of the method is the source code that will be checked. The whole process contains three main phases: the Pipeline Analyzer, the Parser/Deparser Analyzer, and the Final Checker. The Pipeline Analyzer has two parts: the call graph, and the Condition calculator. The Pipeline Analyzer checks all of the components – actions, tables, and control functions – in the P4 source and calculates pre- and postcondition pairs for them while checking their correctness. The Parser/Deparser Analyzer works with the parser – gets the main precondition from it – and the deparser – gets the main postcondition from it. The Final Checker works with the calculated pieces of condition and checks that if every main precondition matches with at least one needed precondition. If the matching is not correct then it means that there are some errors in the code, because after the given parsing – which is described by the main precondition – there is not any execution path, which could work properly. However, if the matching is right in the preconditions then it checks the same with the postconditions.

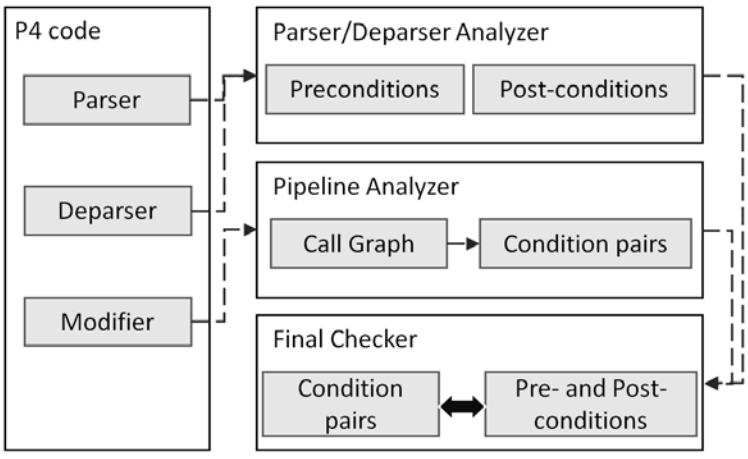

Figure 2: Model of the method

4.1 Pipeline Analyzer

The analyzer consists of a call graph and the Condition Calculator. The call graph contains the relationship of the call between the control functions, tables, and actions. Based on the graph it will make an order of these components to be processed by the Condition Calculator. The Condition Calculator goes through all of them and creates their pairs of pre- and postcondition, which describes the correct working of the program.

This phase processes the modifier part of the P4 code, and calculate pre- and postconditions for the different components. The result of this phase shows the claim – what kind of header information it needs to work well – and the offering – what kind of header information it will create after the processing – of the component.

Call Graph The call graph handles the modifier part of the input. Its vertices are the components of this part – the control functions, match-action tables, and actions – and the directed edges describe a calling relation between them.

The processing of the modifier part, in this version of the method, is the simplest packet processing path, where there is only an ingress pipeline. Therefore there is an initial vertex, which is the main ingress control function. The forwarding steps to process the whole control flow and whenever a component calls a component – if that has not been added to the graph as a vertex, it will create a vertex for it – and will give an edge from the vertex of the calling component and the vertex of the called one.

After the processing of the code, it will create a list of the components. This list starts with those components, which do not depend on any other components, – they have no outgoing edges –, it continues with components, which calls the components, which have already been added to the list, until the last two components, which will be the main function of the ingress. In the call graph there can not be any loop, because of the specification of the P4 language [6], which defines that neither direct recursion nor mutual recursion can be between actions and controls.

Condition calculator The list of the components gives a hint for the Condition Calculator in which order it should process the components, therefore whenever a command of a component's calling is processed, the condition of that component will have been calculated. During the processing of the components, it uses a rule system, which will calculate a *ConditionState*.

$$\begin{aligned}\text{ConditionState} \in \{&\mathit{name} : [(\mathit{pre} : \{\mathit{valid} : [\mathit{ids}], \mathit{invalid} : [\mathit{ids}]\},\\ &\mathit{post} : \{\mathit{valid} : [\mathit{ids}], \mathit{invalid} : [\mathit{ids}]\})]\}\end{aligned}$$

Figure 3: Condition State Describer

The type of the *ConditionState* – Figure 3 – is a set of pairs, where a pair shows the name of the component and its list of pre- and postconditions as pairs.

Every condition consists of a *valid* and an *invalid* container, both of them contain a list of headers' and fields' names. During the method, we use the *valid* word as a synonym of the property of initialized fields.

$$\vdash: (\text{ConditionState} \times \text{Name} \times \text{Program}) \to \text{ConditionState}$$

Figure 4: Type of $\vdash$

The Condition Calculator is defined by the operator $\vdash$, the type of which can be seen in Figure 4. It works with a known *ConditionState*, a name of a component – the one, that is processed when it uses this rule – and the program code. The rules are based on the structure of the program, therefore they have a deterministic usage. In this version of our method, we only work with the main structures of the program, therefore the rules only use a subset of the P4 language.

We define the behavior of the Condition Calculator with a rule system. It uses rewriting rules – we have an expression in the bottom and we rewrite it to the top of the rule. In Figure 5, there are the basic rules, which can be used for the subset of the P4 program. The rules for the assignment, sequence, table, skip, branch, table calling, and validity settings.

The preprocessing of the usage of the rules is to create the initial *ConditionState*. It is an empty state, where all of the components are stored, but all of them have only one condition, which says that the *drop* is invalid in the postcondition. The *drop* is a unique name in the conditions, which shows if the packet is dropped – it is valid, if it is dropped, and invalid if it is not dropped.

In the rules, we use some notations:

- $(A \parallel B)$ shows a rewriting from A to B. If the rewrote element has already been in a set, then it will be deleted from that place, and it will be only in the new set.
- *ids* means the type of the identifications. For example $\{ipv4, ipv4.ttl\}$ is a set of ids i.e. {ids}.
- Ids is a function, which gives the names from an expression. For example the Ids of the "$ipv4.ttl - 1$" is a $\{ipv4, ipv4.ttl\}$, which contains the identification of the header and the field of the expression.
- $newIds$ is a function, which gives a subset of the Ids. These elements have not been used in any condition state.
- $oldIds$ is a function, which gives a subset of the Ids. These elements have already been used in a condition state.
- $fields$ is a set of the identification of the fields of the given expression.
- *header* is a set of the identification of the headers of the given expression.

- $C(conditionState, condition)$ calculates those pairs from the condition state, which fit to the condition.
- $H.n$ shows the pre- and postcondition pairs of the component named n in the Condition State H.
- $Empty(H.n)$ creates an empty block for the new component.
- $V : \{ids\} \rightarrow Bool$; $V(I) = \forall i \in I(H.n.post.Valid(i))$. It checks if every header, field or variable identification is valid in the postconditions of the actual component – the examined *ConditionState* H and component n come from the rule.
- $A \dot{\cup} B ::= ifcond(A.post =:= B.pre, A.pre, A.post \times B.post \uparrow)$ and

  $A \uplus B ::= ifcond(A.pre =:=, A.pre \times B.pre, A.post \times B.post \uparrow)$, where

  * $ifcond(condition, precondition, postcondition)$ - it checks if the given condition is true. If it is true, then it creates a pre- postcondition pair, where the precondition is defined in the second and the postcondition is defined in the third argument.
  * $=:=$ checks if two given conditions fit or not.
  * $\times$ merges the given conditions. If there is an $\uparrow$ in one side of the merging it means to use that condition as a prior one – if there is a header information, which appears in both conditions, the condition with the $\uparrow$ will be stored.

The rules The rules are based on the structure of the program, therefore the calculation is deterministic. On the sides of the rules, we can see two types of possible calculations. In the rules, there is the form $H, n \vdash S$, where H is a *ConditionState*, n is the name of the processed component and S is the *Program* that it processes. In one case, these rules can be recursively rewritten by the same form of expression – like in Rules 2, 6 – or in the other case it can be rewritten with a *ConditionState* – as in Rules 1, 3, 4, 5, 7, 8.

Rule 1 shows that the *Skip* does nothing, therefore the result will be the calculated condition state.

Rule 2 defines the processing of the sequence construct with composition – first, it processes the $S1$ program, and from the result of this calculation it processes the $S2$ program.

Rule 3 describes the assignment. Here we can see a side condition of the rule, which says to check if the identification of the expression *expr* – right side – and the header of the e – left side of the assignment – is valid. The side condition in this usage means, it will continue the calculation with those pairs of conditions, which are correct based on the side condition – in the pairs of conditions, where the side-condition is false, the process shows an error case. If the condition is true

$$1.\ \frac{H}{H, n \vdash Skip}$$

$$2.\ \frac{(H, n \vdash S1) \vdash S2}{H, n \vdash S1; S2}$$

$$3.\ \frac{H[H.n||H.n[pre \to Valid(expr.newIds, e.newIds.header), \\ post \to Valid(e.Ids, expr.newIds)]]}{H, n \vdash e = expr}$$

if $V(expr.oldIds)\&V(e.oldIds.header)$

$$4.\ \frac{H[H.n||H.n[post \to \{Valid(h), Invalid(h.fields)\}]]}{H, n \vdash h.setValid()}$$

$$5.\ \frac{H[H.n||H.n[post \to Invalid(h, h.fields)]]}{H, n \vdash h.setInValid()}$$

$$6.\ \frac{(H[C(H.n, cond)]), n \vdash S1 \cup (H[C(H.n, !cond)]), n \vdash S2}{H, n \vdash \text{if } (cond)\ S1 \text{ else } S2}$$

$$7.\ \frac{H[H.n|| \bigcup_{i=1}^{n} (H.n[pre \to Valid(k.Ids), post \to Valid(k.Ids)] \uplus a_i)]}{H, n \vdash keys : k\ \ actions : \{a_1, ..., a_n\}}$$

$$8.\ \frac{H[H.n||(H.n \dot\cup H.table\_name)]}{H, n \vdash table\_name.apply()}$$

Figure 5: Definition of $\vdash$ in a subset of the language

then it refreshes the condition state of the processed component by giving the new identifications with a valid property to the precondition and sets them valid in the postcondition too – because this command has not changed them – and sets the left side of the assignment to valid too – because it has just got a new value.

Rule 4 and 5 describe the validity setting. These functions set the validity of the header to valid or invalid – depends on the called function *setValid*() or *setInvalid*() – and set all of the fields to uninitialized.

Rule 6 gives the calculation of the branches. It needs to work with two different execution path – one where the condition is true, and one where it is false. If we check the actual condition state, it may contain several pairs of pre- and post-condition, therefore it describes both of the possible executions. In this rule, the calculation divides into two ways, these are calculated separately and their result

will be joint.

Rule 7 calculates the conditions for the tables. It is important for the tables to read valid headers and initialized fields as keys. Although when this rule is used, we process a fresh component, therefore it does not need to use any side-condition, because this condition will be described in the precondition. First, it creates the precondition, which contains the condition of valid keys, and then it uses the precalculated conditions of called actions and merges them into several pairs of conditions. It checks if the preconditions of the actions fit the precondition of the table, and stores the merged conditions – uses the merged preconditions and the merged postconditions, where the postconditions of the actions are the stronger.

Rule 8 describes a table calling. In this case, the conditions for of the table has already been calculated, therefore it only needs to merge them to the actual conditions. In this merge, it has to check if the actual postcondition and the precondition of the table fit, and when it is correct it can merge them – uses the actual preconditions and merge the postconditions, where the postcondition of the table is the stronger.

4.2 Parser/Deparser Analyzer

The Parser/Deparser Analyzer is based on our previous paper [13]. It uses the parser to describe the main preconditions and the deparser to defines the main postconditions of the whole program. Therefore it describes the possible header information of the input and the output packets. The result of this is not pairs of conditions, but sets of pre- and postconditions. These conditions are not connected – are not in pairs –, they describe that what kind of initial states we would like to start our program and which final states are proper for the end.

There can be more main pre- and postconditions because parser and deparser can contain branches. In the main postconditions besides the conditions from the branches, there is another unique condition, which describes the case, when the packet is dropped. In this case, the *drop* is valid, and the validity of the other header information is irrelevant.

4.3 Final Checker

The task of the Final Checker to verify if the expected main pre- and postconditions of the program – which come from the Parser/Deparser Analyzer – and the actual pre- and postconditions fit well to each other. For this calculation we can generate a formula – Figure 6 from the calculated conditions and check if the formula is valid.

$$\forall Pr \in MainPre,\ \forall p \in pipeline\_condition\,,$$
$$\exists Po \in MainPost:\ ((Pr \supset p.pre) \wedge (p.post \supset Po))$$

Figure 6: Formula for verification

It checks if the program from every initial state – which are states, where the precondition is true – there is a final state – which are states, where the postcondition is true – to reach. Therefore it checks for all precondition if they have at least one postcondition to fit based on the condition of the components. These components describe the pre- and postconditions for the correct working of the component. It checks, if any of the preconditions of the biggest component – in this version, it is the main ingress control function – is correct based on the main precondition. If there is one, then it checks the postcondition of it and checks the correctness of the actual postcondition.

If there is an actual precondition, which does not fit the main precondition, it means, there is an execution path, which will not work properly. If there is an actual precondition, which fits a main precondition, but the postconditions are not fitting, it means, during the calculation of that execution path, there will not be any problem, but the final state will not be correct for the specification of the program – in this case, the deparser of the program.

4.4 Error detection cases

The method can detect error cases in four different ways.

Condition Calculator It can detect errors, those are caused by the code of a component or can detect if the components are not matching. Three examples can be seen in Figure 7.

In the left one, *act1* and *act2* contains inside errors, which come from the bad usage of the method *setValid*() and *setInvalid*(). The side-effect of both methods is to change the uninitialized fields of the header. Therefore in the first case, when it would like to read the right side of the assignment, it will get a valid header and uninitialized field, so the read value is unknown. In the second case, on the left side of the assignment, it would like to write a field of an invalid header, which is an error case too. Both of them will be detected when it uses Rule 3, when the side-condition will be false in the process of the assignment.

On the right side of Figure 7, there is an example of the components, which are not matching. There are two components: the table t and a control function. The precondition of the table t is: $Valid(ipv4, ipv4.dstAddr)$, because it reads the *ipv4.dstAddr* field, so it need a valid header and initialized field. Although the true case of the branch says that the ipv4 is invalid – the postcondition of the control function stores it. Therefore when it uses Rule 8, it will not be able to merge the conditions, because the precondition of the table and the postcondition of the control function will not fit.

Final Checker The other case, when errors can be detected, is the process of the Final Checker. It can detect if the specification of the program does not fit the actual conditions i.e. starting the program from any initial state – where the main

```
action act1 () {
   hdr.ipv4.setValid();
   hdr.ipv4.srcAddr = hdr.ipv4.dstAddr();
}

action act2 () {
   hdr.ipv4.setInValid();
   hdr.ipv4.srcAddr = hdr.ethernet.dstAddr();
}
```

```
table t {
 key = { hdr.ipv4.dstAddr : exact; }
 ...
}

apply{
 if (!hdr.ipv4.isValid(()) {
   t.apply() }
}
```

Figure 7: Error cases in Condition Calculation

precondition is true – it can reach one of the final states – where the postcondition is true. In this phase, there can be two types of errors.

The first case is when the preconditions – of the main and the actual conditions – are matching, but there is no main postcondition, which fits the actual postcondition. In this case, although the program can start the execution from a good state, it will reach a state, which is not expected by the deparser – the calculated header information will be not correct for the program.

The second case is when there is no matching actual precondition for the main precondition. This means that starting from that input header information, the program will reach an error, the execution will be incorrect.

5 Case study

Figure 1 is an example P4 program which will be used to show an example usage of the method.

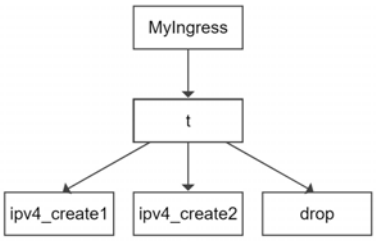

Figure 8: Call graph of the example program

Pipeline Analyzer First it creates the call graph. The graph of the example can be seen in Figure 8. There is only an ingress pipeline – *MyIngress* –, which calls one table – *t*– , which can call one of the three actions – *ipv4_create1*, *ipv4_create2* and *drop*. Based on the graph it can create the following list to define the order of the processing:
[*ipv4_create1*, *ipv4_create2*, *drop*, *t*, *MyIngress*].

```
{ MyIngress: [(
      pre:  { valid: [], invalid: []},
      post: { valid: [], invalid: [drop]})],
t: [(
      pre:  { valid: [], invalid: []},
      post: { valid: [], invalid: [drop]})],
ipv4_create1: [(
      pre:  { valid: [], invalid: []},
      post: { valid: [], invalid: [drop]})],
ipv4_create2: [(
      pre:  { valid: [], invalid: []},
      post: { valid: [], invalid: [drop]})],
drop: [(
      pre:  { valid: [], invalid: []},
      post: { valid: [], invalid: [drop]})]}
```

Figure 9: Initial ConditionState

Based on the order of the list, the Condition Calculator starts the calculation from the expression, which is in Figure 10, where *Empty* is an initial *ConditionState*, which can be seen in Figure 9, which is mentioned in section 4.1.

(((Empty, "ipv4_create1" ⊢ ipv4_create1), "ipv4_create2" ⊢ ipv4_create2),
"drop" ⊢ drop), "t" ⊢ t), "MyIngress" ⊢ MyIngress

Figure 10: Initial expression

Condition calculator The action *ipv4_create1* consists of sequences – Rule 2 – of two assignments – Rule 3 – and one setting of validity – Rule 4. Therefore there will be three rewritings of the conditions. When it starts the processing, there is only one pre- and postcondition pair, where the only condition says that the *drop* is invalid. The process of the first assignment – $hdr.ipv4.srcAddr = hdr.ethernet.dstAddr$ – does not contain any side-condition, because it has just started the calculation. Therefore it will add the *ethernet*, *ethernet.dstAddr* and *ipv4* to the Valid conditions in both of the pre- and postcondition – because it would read the right side, and it can only give value to a field of a valid header in the left side of the assignment. During the calculation, the postcondition describes the actual state. Therefore we will try to continue the process with the second assignment – $hdr.ipv4.dstAddr = dstAddr$. The side-condition checks, if the header of the written field is valid – *ipv4* is valid in the actual state or in the postcondition –, and if the read expression is valid – *dstAddr* is the parameter of the action, therefore the side-condition is not working with it. The validity is correct, so it adds the *ipv4.dstAddr* to the Valid side of the postcondition – because after the assignment, its value will be initialized. After the assignments, the program tries to set the validity of the header *ipv4* to valid. It changes only the postcondition by

make all of the fields of *ipv4* – *ipv4.srcAddr*, *ipv4.dstAddr* and *ipv4.ttl* – invalid.

The calculation of the other actions similar to it, their results can be seen in Figure 11.

```
ipv4_create1: [(
      pre: {
        valid: [ethernet, ethernet.dstAddr, ipv4],
          invalid: []},
      post: {
        valid: [ipv4, ethernet, ethernet.dstAddr],
        invalid: [ipv4.ttl, ipv4.srcAddr, ipv4.dstAddr, drop]})],
ipv4_create2: [(
      pre: {
        valid: [ethernet, ethernet.dstAddr],
        invalid: []},
      post: {
        valid: [ipv4, ipv4.srcAddr, ipv4.dstAddr,
                ethernet, ethernet.dstAddr, drop],
        invalid: [ipv4.ttl]})],
drop: [(
      pre: {
        valid: [ethernet, ethernet.dstAddr],
        invalid: []},
      post: {
        valid: [drop, ethernet, ethernet.dstAddr],
        invalid: []})],
```

Figure 11: Conditions of the actions

After the actions, it processes the table *t*. It has one key, so first, it sets its header *ethernet* and field *ethernet.dstAddr* to Valid, as a precondition, because the program would like to read them – and of course, as a postcondition too, because that shows the actual condition. Then it checks and merges its conditions and the conditions of the actions together. Therefore there will be three pairs of conditions – because of the three actions. The precondition of action *ipv4_create1* matches with the newly defined precondition of the table, therefore it can create the merged pre- and postcondition, where the condition of the action is the stronger one – in this case there is no conflict, so the condition pair of the action will be one of the table conditions.

The process of the other actions is similar. Their preconditions fit the precondition of the table, therefore it can merge their conditions. The conditions of the ingress *MyIngress* are the same as the table because it just simply calls it. The conditions of the table and ingress pipeline can be seen in Figure 12.

Parser/Deparser Analyzer The specification of the program is calculated from the parser and deparser. Both the parser and the deparser are really simple. None of them contains any branch, therefore there is only one possible precondition and two postconditions – Figure 13.

```
1.  [(
2.       pre: {
3.         valid: [ethernet, ethernet.dstAddr, ipv4],
4.         invalid: []},
5.       post: {
6.         valid: [ipv4, ethernet, ethernet.dstAddr],
7.         invalid: [ipv4.ttl, ipv4.srcAddr, ipv4.dstAddr, drop]}),(
8.       pre: {
9.         valid: [ethernet, ethernet.dstAddr],
10.        invalid: []},
11.      post: {
12.        valid: [ipv4, ipv4.srcAddr, ipv4.dstAddr,
13.                ethernet, ethernet.dstAddr],
14.        invalid: [ipv4.ttl, drop]}),(
15.      pre: {
16.        valid: [ethernet, ethernet.dstAddr],
17.        invalid: []},
18.      post: {
19.        valid: [ethernet, ethernet.dstAddr, drop],
20.        invalid: []})]
```

Figure 12: Conditions of table t and ingress pipeline *MyIngress*

The precondition describes the case when the parser extracts only the *ethernet* header, therefore this header and all of its fields will be valid, and the other header *ipv4* and its fields will be invalid.

There can be two postconditions. One, which is defined by the deparser, where the packet is not dropped and both of the headers *ethernet* and *ipv4* and all of their fields are valid – because the program will use their value for the new packet. The other case is when the packet is dropped.

```
Pre: [{
  Valid: [ethernet, ethernet.allFields() ],
  InValid: [ipv4, ipv4.allFields(), drop ]}]

Post: [{
  Valid: [ethernet, ethernet.allFields(), ipv4, ipv4.allFields() ],
  InValid: [drop] },

  { Valid: [drop], InValid: [] }]
```

Figure 13: Specification of the program

Final Checker There is only one main precondition, therefore it needs to check if the possible actual preconditions – the preconditions of the ingress pipeline *MyIngress* – fit, then their postcondition fits one of the main postconditions.

The main precondition assures only valid *ethernet* header information, although the first condition pair – rows 2-7. Figure 12 – of the actual condition needs a valid

ipv4 header too. The first part of the formula is false, therefore the method can detect an error, which shows that this execution path of the program will not be able to execute well, because of the invalid *ipv4* header.

In the second condition pair – rows 8-14. Figure 12–, the precondition fit to the main precondition, therefore it needs to check if the actual postcondition assures one of the main postconditions. The actual condition describes an execution when the packet was not dropped, therefore only the first main postcondition can be good, but we can see that in the actual postcondition the *ipv4.ttl* field is invalid, but the main postcondition needs a valid one. It is another error case, which is detected and it shows the case when the execution is correct, but the final header information which is calculated does not match with the expected result by the deparser.

In the third condition pair – rows 15-20. Figure 12, the preconditions fit, and this is the case when the packet was dropped during the execution, therefore its postcondition will ensure the second main postcondition. This path shows a correct execution path.

6 Evaluation

We created a prototype for the Condition Calculator. The critical part of the implementation is the calls of the tables because this is the step when more and more condition states can be defined. Therefore Figure 14 shows the runtime based on different numbers of table calls. During the measurement only one table was used, which has 5 action calls - one *NoAction*, which is like a *skip* program; one *drop* action, which only signs the package to drop it; and three others, all of them contain sequences of simple assignments.

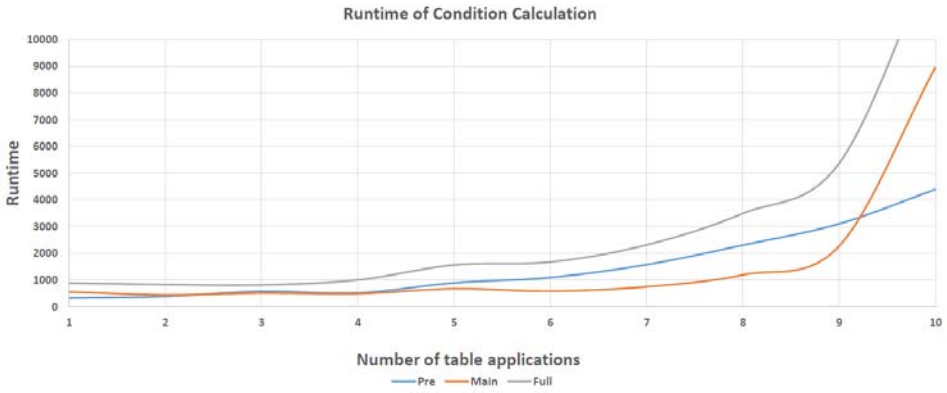

Figure 14: Runtime based on different number of table calls

In the graph of Figure 14, the vertical information is the number of table applications, and the horizontal data is the runtime in milliseconds. We can see three

different lines because the implementation of the calculator is separated into two parts: a pre-calculator, which only prepares the obtained information; and the main condition calculator, which uses the prepared data to create the conditions. In the graph, the three different lines show the runtime of the pre-calculator (blue), the main calculator (orange), and the full time of them (grey). This graph shows the worst cases when all of the calculated conditions are different, therefore the number of the conditions is equal to the number of the actions to the power of the number of the table applications i.e. in our table there were 5 actions, and when we applied this table five times, then we could get 3125 conditions. However, we can see that, when we used 10 table applications, the runtime was only 13 seconds.

This presented data only give the worst runtime of this calculation, but when we use a normal P4 code then most of the conditions are the same, do the program will not work with this number of conditions.

7 Summary

In this paper, we introduced a formal method to detect errors in P4 programs by checking the pre- and postconditions. It has three main analyzers: the Parser/Deparser Analyzer, the Pipeline Analyzer, and the Final Checker. The method only works with the P4 source, therefore it extracts the specification of the program from the parser and deparser of it – by the Parser/Deparser Analyzer. The Pipeline Analyzer checks the definition of the pipeline, and calculated condition pairs for the different components of it – actions, match-action tables, and control functions. It can detect four different types of errors, which can be caused by the incorrect usage of invalid headers and uninitialized fields or components, which do not match.

Future Work The method is a base of a full solution for P4 verification. Until this goal, we need to extend the checked subset of the P4 language, therefore we will be able to work with more programs.
When both of our approach is ready – this, component-based and our previous idea [13] –, then it could be an interesting work to check which one is better in the specified error detections, and combine them for a more optimized solution.
P4 programs can use different architectures. In this approach, we only showed the Protocol Independent Switch Architecture, which use a single pipeline, but there are other architecture models – for example Portable Switch Architecture [3] which can work with more pipelines. Our solution will be able to used as a part of a research, which checks if a P4 program is correct for a given architecture, while checking the correctness of a pipeline.

Acknowledgement

We thank Dániel Lukács (PhD Student at ELTE) for the preparation of the base of the tool, therefore we could integrate our implementation of this verification idea into it!

References

[1] $P4_{16}$ Language Specification. `https://p4.org/p4-spec/docs/P4-16-v1.1.0-spec.pdf`, 2018. Online; accessed 7 December 2020.

[2] The P4 Language Specification. `https://p4.org/p4-spec/p4-14/v1.0.5/tex/p4.pdf`, 2018. Online; accessed 7 December 2020.

[3] $P4_{16}$ Portable Switch Architecture (PSA). `https://p4.org/p4-spec/docs/PSA.html`, 2020. Online; accessed 7 December 2020.

[4] Birnfeld, K., da Silva, D. C., Cordeiro, W., and de França, B. B. N. P4 switch code data flow analysis: Towards stronger verification of forwarding plane software. In *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, pages 1–8, 2020. DOI: `10.1109/NOMS47738.2020.9110307`.

[5] Bosshart, Pat, Daly, Dan, Gibb, Glen, Izzard, Martin, McKeown, Nick, Rexford, Jennifer, Schlesinger, Cole, Talayco, Dan, Vahdat, Amin, Varghese, George, and Walker, David. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, 2014. DOI: `10.1145/2656877.2656890`.

[6] Budiu, Mihai and Dodd, Chris. The p416 programming language. *SIGOPS Oper. Syst. Rev.*, 51(1):5–14, September 2017. DOI: `10.1145/3139645.3139648`.

[7] Eichholz, Matthias, Campbell, Eric, Foster, Nate, Salvaneschi, Guido, and Mezini, Mira. How to avoid making a billion-dollar mistake: Type-safe data plane programming with safep4. *CoRR*, abs/1906.07223, 2019.

[8] Freire, Lucas, Neves, Miguel, Leal, Lucas, Levchenko, Kirill, Schaeffer-Filho, Alberto, and Barcellos, Marinho. Uncovering Bugs in P4 Programs with Assertion-based Verification. In *Proceedings of the Symposium on SDN Research*, SOSR '18, pages 4:1–4:7, New York, NY, USA, 2018. ACM. DOI: `10.1145/3185467.3185499`.

[9] Liu, Jed, Hallahan, William, Schlesinger, Cole, Sharif, Milad, Lee, Jeongkeun, Soulé, Robert, Wang, Han, Caşcaval, Călin, McKeown, Nick, and Foster, Nate. P4v: Practical Verification for Programmable Data Planes. In *Proceedings of*

the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM '18, pages 490–503, New York, NY, USA, 2018. ACM. DOI: 10.1145/3230543.3230582.

[10] Shukla, Apoorv, Hudemann, Kevin Nico, Hecker, Artur, and Schmid, Stefan. Runtime verification of p4 switches with reinforcement learning. In *Proceedings of the 2019 Workshop on Network Meets AI & ML*, NetAI'19, page 1–7, New York, NY, USA, 2019. Association for Computing Machinery. DOI: 10.1145/3341216.3342206.

[11] Stoenescu, Radu, Dumitrescu, Dragos, Popovici, Matei, Negreanu, Lorina, and Raiciu, Costin. Debugging P4 Programs with Vera. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 518–532, New York, NY, USA, 2018. ACM. DOI: 10.1145/3230543.3230548.

[12] Stoenescu, Radu, Popovici, Matei, Negreanu, Lorina, and Raiciu, Costin. Symnet: Scalable Symbolic Execution for Modern Networks. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 314–327, New York, NY, USA, 2016. ACM. DOI: 10.1145/2934872.2934881.

[13] Tóth, Gabriella and Tejfel, Máté. A formal method to detect possible p4 specific errors. In *Position Papers of the 2019 Federated Conference on Computer Science and Information Systems*, volume 19 of *Annals of Computer Science and Information Systems*, pages 49–56. PTI, 2019. DOI: 10.15439/2019F355.