# Modular Code Parser for the P4 Language

Mohsen Rahmati, François-Raymond Boyer, Bill Pontikakis, Jean-Pierre David, *and* Yvon Savaria
*Polytechnique Montréal*

## Abstract

We present an extension over the open-source P4C compiler, for modular header parsers for the P4 language. Modularity is essential to obtain code reusability, composability, and incremental programming. The modular parser includes matching and resolving names of the identifiers of two parser graphs, comparing them, and finally merging them. A significant feature of this modular parser is the backward compatibility of pre-existing P4 codes. It permits merging parsing codes automatically and allows to offer vendor-customer compatibility without having to learn new syntax or annotations.

## 1 Introduction to the Modular P4 Code Parser

P4 [1] introduces a novel method for enhancing data plane programmability of network devices, but it, unfortunately, lacks good support for modularity, which is crucial for at least three distinct reasons: reusability, composability, and vendor/customer partitioning.

To partly address this challenge, we present a modular code parser for the P4 language that implements three new passes in the front end for the open-source P4 Compiler [2].

Resolving names: We implemented a compilation pass that matches and select a common name for “equivalent” identifiers in both parsers.

Compare two parser graphs: We implemented a second compilation pass that compares two parser graphs state by state to match “equivalent” states.

Merging two parser graphs: We implemented a third compilation pass that merges the two parser graphs according to the result of the other passes.

Testing Modular Parser: We tested the modular parser with simple (P4 tutorial exercises [3]) and more advanced (SRV6 Cisco [4]) examples. The modular code parser also permits to #include some ‘vendor’s’ code in a ‘customer’s’ code without adding new syntax and/or annotation.

## 2 Methodology for Implementing the Proposed Modular P4 Code Parser

In the first step, we match and resolve the names of the inputs and outputs of two parser graphs. We change the names of the inputs and outputs of the second parser graphs to match the first graph for packets, local metadatas, standard metadatas, and headers.

We compare finite-state machines (FSM) of the two parser graphs state by state. The challenge is that programmers can use different names for the same state, so we used protocol numbers for comparing the states of the FSMs. We considered three different scenarios. In the first scenario, we have equivalent states with equivalent children in the vendor and customer code, so we keep only one of the equivalent states. In the next scenario, we have some states in the customer code which are not in the vendor code, so these states are all added separately to the final parser. In the next one, we have some equivalent states with different children, so we need to merge transitions that are not in the vendor state and merge parserLocalElements in the vendor code.

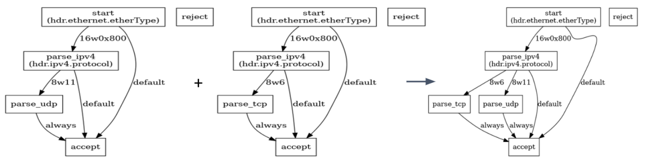

Figure 1: The graphs of UDP, TCP, merged parser using [2].

We may need to merge states with different names, but with the same ”protocol number” (value of the field used for the transition). Another problem is that we may have the protocol number of a state in the parent of that state in the P4 language. For finding the protocol number of each state, we check the name of the state in its parent and compare that name with the name of that state to find the protocol number in each state. Then, we could compare two parser graphs with the protocol numbers except for start, reject, verify, accept, and "Ethernet" (the top header parsed). We use the state name for start, reject, verify, and accept (as defined by the P4 spec), and the top state without protocol number is considered equivalent in both graphs (Ethernet in most examples).

In the case we have corresponding nodes in the vendor’s and customer’s codes, we need to understand which children are not the same and merge their transitions and parserLocalElements, except for extracting packets. Indeed, if we have nodes extracting packets in the first parser code, we do not need to merge extracted packets of the second parser code.

## References

[1] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. 44:87–95, 2014.

[2] "the open-source p4 compiler". https://github.com/p4lang/p4c.

[3] "the p4 tutorials". https://github.com/knetsolutions/p4-tutorials/tree/master/exercises.

[4] "the srv6 network programming". https://github.com/netgroup/p4-srv6/blob/master/p4src/include/parser.p4.