Pyramis: A Domain Specific Language for Developing Multi-tier Systems

Ashwin Kumar
ashkumar@cse.iitb.ac.in
Indian Institute of Technology
Bombay
India

Ajinkya Tanksale
aatanksale@iitb.ac.in
Indian Institute of Technology
Bombay
India

Armaan Chowfin
f20190289@goa.bits-pilani.ac.in
BITS Goa
India

Mohan Rajasekhar Ajjampudi
rajasekhar@cse.iitb.ac.in
Indian Institute of Technology
Bombay
India

Arnav Mishra
arnav@cse.iitb.ac.in
Indian Institute of Technology
Bombay
India

Abuhujair Khan
abuhujairkhan@cse.iitb.ac.in
Indian Institute of Technology
Bombay
India

Vishal Saha
vishalvsaha@cse.iitb.ac.in
Indian Institute of Technology
Bombay
India

Priyanka Naik
priyanka.naik@ibm.com
IBM Research
India

Mythili Vutukuru
mythili@cse.iitb.ac.in
Indian Institute of Technology
Bombay
India

ABSTRACT

Text-based specifications are the de-facto standard for specifying complex multi-tier systems. For example, 3GPP specifications define various interfaces, messages, and message processing at the multiple inter-connected nodes of a 5G system. These standards documents tend to be verbose, and may be ambiguous or inconsistent in places, increasing programmer effort to implement them in a general purpose language. This paper presents Pyramis, a Domain Specific Language (DSL) with suitable high-level abstractions for specifying the interfaces, messages, and processing in a multi-tier system. Pyramis allows programmers to specify multi-tier systems in a concise and precise manner, and enables easy development of software based on the specifications. We also develop a translator with Pyramis that automatically generates optimized, multi-threaded C++ code for the various components of the multi-tier system from the specification, and also generates eBPF-based measurement code for computing various performance metrics. We use Pyramis to build several components in the 5G mobile packet core. We show that the specifications written in Pyramis are 2–3× smaller than the actual reference implementation, while the auto-generated C++ code performs on par with a hand-optimized implementation. We believe that Pyramis can eventually replace verbose text specifications like the 3GPP standards documents in telecom systems.

APNet 2024, August 3–4, 2024, Sydney, Australia

ACM ISBN 979-8-4007-1758-1/24/08
https://doi.org/10.1145/3663408.3663431

CCS CONCEPTS

• **Networks → Mobile networks**; • **Software and its engineering → Specification languages**.

KEYWORDS

5G core, Cellular networks, Domain Specific Language

ACM Reference Format:
Ashwin Kumar, Ajinkya Tanksale, Armaan Chowfin, Mohan Rajasekhar Ajjampudi, Arnav Mishra, Abuhujair Khan, Vishal Saha, Priyanka Naik, and Mythili Vutukuru. 2024. Pyramis: A Domain Specific Language for Developing Multi-tier Systems. In *The 8th Asia-Pacific Workshop on Networking (APNet 2024), August 3–4, 2024, Sydney, Australia.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3663408.3663431

1 INTRODUCTION

Modular software is made up of different components that are functionally interlinked with each other. In a multi-tier system, these components (or nodes) are physically separated into different tiers, and each node is assigned a specific functionality. For the various components of a multi-tier system to inter-operate seamlessly, all nodes in the system should adhere to a common *specification*, which specifies various aspects, e.g., the network connection parameters of each interface exposed by every node in the system to receive messages on (e.g., transport protocol, server port number), message formats for the requests and responses exchanged between nodes on each interface, and how to process each request in order to generate a response.

An example of a multi-tier system, that serves as a compelling motivation for our work, is the 5G mobile packet core [9]. The packet core connects the wireless radio access network to the Internet and other external networks, and serves several critical functions in a mobile network, like registering and authenticating mobile users, managing data sessions, and handling the mobility of users. The 5G packet core is composed of several software network

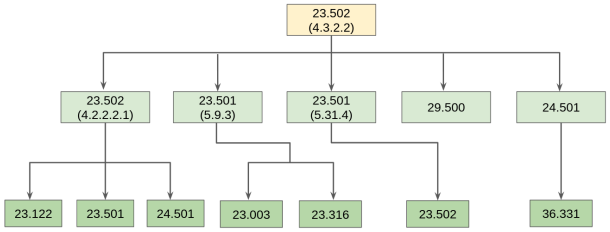


Figure 1: 3GPP Spec Documents Reference Tree

functions (NFs) running on commodity hardware, in accordance with the principle of Network Function Virtualization (NFV) [33]. Each NF in the 5G core handles a separate functionality, and the various NFs communicate with each other over well-defined interfaces to handle any given signaling procedure of a mobile user, e.g., to authenticate a user.

Text specifications are the de-facto standard for specifying complex multi-tier systems, but these specifications tend to be very verbose in nature. For example, the information required to build individual NFs in the 5G packet core is distributed among tens of different documents maintained by 3GPP [3–9], and each of these documents run into several hundreds of pages. Figure 1 shows the 3GPP documents that need to be referred to in order to understand just one of the many steps in the session establishment callflow, that is initiated when a user establishes a data session with the mobile network. Specification document 23.502 [8] describes the message to be exchanged between two nodes at that particular step in the callflow, but details on how to fill the different fields of that message are spewed across several other documents as shown in the figure. These specifications are also revised every few months for every new release, and the large text documents are very hard to maintain in a consistent state across changes in each release. The sheer scale of these documents are not only hard for the specification writers to keep track of, but are also a burden on the programmer assigned with the task of implementing these nodes in a general purpose language (GPL) code.

Text-based specifications can also be ambiguous in nature. Our experience with building a few components of the 5G core has exposed us to several such ambiguities. For example, in the 5G packet core, a user must set up multiple resources in the data plane before sending the actual data [2]. The specification states that a failure message should be sent in the case of "failure of user plane resource setup", but fails to mention which user plane resource(s) is it talking about, leaving the statement ambiguous. Badly written text-based specifications also tend to specify systems which are hard to implement efficiently, because the authors are only writing text and not actual code. The 5G packet core specifications for some request-response message pairs do not include an identifier of the request in their corresponding response messages. In such a scenario, every request/response exchange must be performed on a separate transport layer connection, and multiple requests cannot be multiplexed on the same connection, which leads to an inefficient implementation. The programmers can choose to overcome these obstacles by resolving such ambiguities in their implementations with some local decisions. However, doing so makes implementations across different vendors incompatible, and requires significant testing and debugging effort.

Our work addresses the problem of writing concise and precise specifications for large, complex multi-tier systems that are at a higher level of abstraction than a reference implementation, but lend themselves easily to an efficient implementation. To this end, we seek to develop a DSL to specify multi-tier systems that can bridge the gap between non-programmer domain experts specifying the system, and the programmers implementing it. We introduce Pyramis, a DSL to specify multi-tier networked systems. Pyramis allows the system developer to describe the architecture of the multi-tier system in terms of the interfaces at every node, messages exchanged between nodes, and the processing done at every node, using high-level abstractions and keywords. Pyramis also includes a translator which converts Pyramis code to various pieces of software that can be derived from the specifications. Our translator uses Pyramis specifications to generate efficient multicore-scalable C++ code to realize a working implementation of all the components in the multi-tier system. The translator also allows users to generate eBPF-based measurement code that calculates various node-level performance metrics from the message format specifications.

Previous work on DSLs for networking systems mainly consists of DSLs that specify network protocol stack processing at a single node [13, 20, 23, 24, 27], and DSLs that give the programmer the ability to specify and generate parsers for L7 application protocols [26, 28, 29, 31], both of which are not sufficient to completely specify a complex multi-tier system. While the motivation for our work has been to develop a better way of specifying complex multi-tier systems like the 5G packet core, we believe that our language is extensible to specify multi-tier systems in any domain.

We evaluate the benefits of Pyramis by developing complex NFs in the 5G packet core using Pyramis. Our results when comparing Pyramis code with that of an optimized production-grade implementation of the 5G core [1] written without Pyramis show that Pyramis code has at least 2–3× fewer lines of code than the baseline implementation, while the performance of the translated C++ code auto-generated by Pyramis is on par with that of the baseline. We also demonstrate the correctness of Pyramis generated measurement code. Pyramis is completely open-source, and language description along with the translator code can be found at https://github.com/synerg5g/Pyramis.

2 LANGUAGE DESIGN

We design the Pyramis language keeping the following objectives in mind. **(i) The language should have suitable abstractions to completely capture multi-tier systems like the 5G packet core.** A complete definition of a multi-tier system contains information about the nodes in the system, the interfaces each of these nodes expose to receive messages on, the format of each message flowing through every node in the system, and the processing to be done at each node on receiving a certain message (request). We would like our specifications to be more precise than text-based specifications (which can be ambiguous at times), and more concise than a reference implementation in a GPL. **(ii) The language should enable auto-generation of software based on the specification.** Multi-tier system specifications are used by developers to write code for the system components themselves, as well as auxiliary supporting code for various measurement and monitoring tasks in the multi-tier system. We aim to develop a translator for

```
{
  "Node_A": {"Interfaces": [...]},
  "Node_B": {
    "Interfaces": [{
      "Name": "I2",
      "Protocol": "TCP",
      "Port": 2222,
      "Processing": ["http_processing"]}]
  },
  "Node_C": {"Interfaces": [...]}
}
```

Listing 1: Interface File

```
EVENT http_processing (request)
  DECODE(decode_request_A, message, request)
  IF (message.header.uri == "/requestA")
    CALL (request_from_A, message)
  ...

EVENT request_from_A (message)
  STORE (contextMap, message.key, field2, message.fieldy)
  STORE (contextMap, message.key, field1, message.fieldz)

  CREATE_MESSAGE (newMsg, type_response_A)
  UDF (fetch_value, value, message.fieldx)
  SET (newMsg.field1, value)
  ...
  ENCODE(encode_response_A, msg_response_A, newMsg)
  SEND (msg_response_A, "8.9.10.11", NODE_B, NODE_A,
     InterfaceA)
```

Listing 2: Processing File

Pyramis which generates C++ code for NF implementations, as well as eBPF [19] measurement code, from Pyramis specifications. The translator should perform various syntax checks on the input Pyramis code, and should automatically generate high-performance software from the specifications. **(iii) The language should be extensible.** It should be easy to add new abstractions or primitives to Pyramis to extend the language to use cases that we have not foreseen. We also aim to provide abstractions that enable programmers to insert custom C++ code either to describe parts of the system that are hard to specify using Pyramis abstractions or to insert legacy code that the developer wishes to reuse.

We now describe the design of Pyramis that satisfies the above goals. The programmer writes specifications in Pyramis in the following files using the abstractions (or keywords) provided by Pyramis. The first three of these files (interfaces, messages, and user-defined functions) are common to all nodes, while the last two (processing and query) must be written separately for each node in the system, to generate the processing code and measurement code at each node.

(1) Interface specification: Listing §1 shows a sample interface specification written in Pyramis. The Pyramis programmer lists out nodes, and the interfaces at each of the nodes, in the interface file written in JSON. A node definition consists of a list of interface definitions that it exposes. Each interface definition consists of an identifier for the interface, the transport layer protocol and the port number on which the node expects to receive messages on, and the processing event to be triggered on receiving a message on said interface.

```
# filter on request URI
FILTER(http_header.uri,"/ueAuth/security")
# filter on value inside a field in message
FILTER(message.field_X, value_X)
# groupby on a value inside a field in message
# and count number of messages in each group
GROUPBY(message.field_Y, [COUNT()])
```

Listing 3: Query File

(2) Message specification: All messages exchanged between nodes are listed in the message file by the Pyramis programmer. Message specifications should list out every field in a message, and the data type of the corresponding field. The Pyramis programmer is also expected to provide C++ functions to encode (i.e., convert a message structure into a byte stream) and decode (i.e., convert a byte stream buffer into message structure) each message.

(3) User Defined Functions: Keeping the extensibility and reusability goals in mind, Pyramis allows programmers to write user defined functions in C++, which can be called from the message processing code written in Pyramis.

(4) Processing specification: A key part of Pyramis specification is the processing file that specifies the processing performed at each node upon receiving a message. Listing 2 shows a snippet of the processing file at a node B that receives a HTTP request from a certain node A, and sends a response back to A. All message processing code in Pyramis is written in event blocks using the *EVENT* keyword. Events are triggered either on receiving a message (*http_processing* event in the listing), or from another event inside the processing file using the *CALL* keyword. The *DECODE* and *ENCODE* keywords call C++ functions written by the Pyramis programmer as part of the message specifications to decode/encode a message. The *STORE* keyword is used to store a value against a key in a C++ hashmap, and the complementary *LOOKUP* keyword fetches values from that map. These two keywords together allow programmers to maintain application state. The *UDF* keyword is used to call user-defined functions written by the Pyramis programmer in C++. The *CREATE_MESSAGE* keyword instantiates a message definition whose type is defined in the message file. The *SET* keyword is used to set values of various fields in a particular message. Finally, the *SEND* keyword is used to send a message to another node in the multi-tier system, optionally specifying a callback event when a response is received. The language also has keywords like conditional if/else statements, loops, and allows for statements to be written in a Python-like syntax. We omit a detailed discussion on the complete language syntax due to space constraints. However, we would like to mention that the language is complete enough to handle the processing of various messages in several signaling callflows of the 5G packet core.

(5) Query specification: Pyramis specifications can be used for several purposes besides just specifying message processing in the system. To showcase the power of writing specifications in a

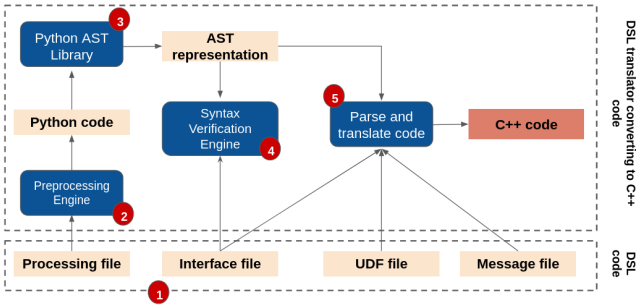


Figure 2: Translator Design

DSL, we add a few extra keywords to Pyramis specifications that allow developers to write node-level queries (see listing 3) to gather meaningful metrics from requests and responses seen at a node. The query specification builds on the message structure in the message file, and uses various SQL-like keywords to extract various metrics from the fields of a message. For example, the *FILTER* keyword is used to identify a subset of messages to process for metric extraction. The *COUNT* keyword counts messages that match the filter, whereas the *STORE* keyword extracts the value of a field from the message and stores that value in a hashmap. The *GROUPBY* keyword is used to group messages by the value of a certain field, and then perform other operations (filter, count or store) on each of those groups. We want to emphasize that the list of keywords provided to write the query specification is not exhaustive, but only serves to show that the Pyramis specification of the system can also be used for auxiliary monitoring and measurement tasks with minimal effort.

3 TRANSLATOR DESIGN

Given a Pyramis specification, the Pyramis translator auto-generates various pieces of code that would have been manually written by developers from a text-based specification. In order to keep the Pyramis specification lighter than a complete implementation, users need not specify complete data types in the Pyramis specification, and the translator auto-infers data types of variables used in the Pyramis code to generate complete C++ code. Using the insight that all variables used during application processing at nodes originate from some message received at the node, the translator uses information about the variables embedded in the message formats to infer their data types. Using a translator instead of manually writing NF code has two advantages. First, the translator ensures that the specification is consistent, error-free, and implementable without ambiguities. The translator performs various syntax and semantic checks on the specification, and throws errors at compile-time in case the specification has bugs. Second, the translator allows generation of optimized high-performance code with minimal programmer effort, since the performance optimizations can be baked into the translator logic, and need not be performed by each system developer separately.

Figure 2 shows overview of the Pyramis translator that generates C++ code for the various components of the system. ① The programmer writes Pyramis specification in the specific files using the abstractions (or keywords) provided by Pyramis (see section §2). ② The processing file first passes through a pre-processing engine which converts code written in Pyramis to Python-like syntax.

Use Cases	LOC		LOC factor
	Pyramis code	GPL code	
AMF (Initial Registration)	711	2307	3.24
SMF (Session Establishment)	891	2362	2.65

Table 1: Lines of Code (LoC) Comparison: Pyramis specifications vs. non-Pyramis C++ implementations

③ We pass this Python code through the Python AST library to generate an Abstract Syntax Tree (AST) representation of Pyramis code. The AST is an intermediate representation of the source code in a tree-like structure that is generated by compilers during the compilation phase. It is useful in semantic analysis and correctness checking of the code. We reuse the Python AST library to build our translator in order to avoid reinventing the wheel in developing our own translator from scratch. ④ The AST intermediate representation along with the interface file pass through a syntax verification engine which detects any syntax errors in the specification, e.g., in the names of interfaces or event handling functions invoked. Finally, ⑤ the Pyramis translator parses the Pyramis code by traversing the AST, and translates it to C++ code. In this step, the translator uses information from the interfaces, messages, and UDF files as well to generate the complete reference implementation. The interface file is used by the translator to auto-generate the code for communication between nodes using standard network primitives like sockets. The auto-generated code uses multi-threading to achieve higher performance on multicore systems.

We use a similar translator design to generate code that extracts performance metrics from the requests and responses seen at the node. While this code could have been generated as part of the application code itself, we choose to generate eBPF code to collect performance metrics, in order to integrate our measurement module with other eBPF based network telemetry systems in the future [15, 17, 18, 21, 30]. The translator autogenerates eBPF code that runs at the TC [12] hook as it can inspect both ingress and egress messages. Each query keyword can be viewed in the same breadth as an operation (filter, count, groupby), wherein the specified field of a particular message is the operand on which the operation is to be performed. Pyramis translator parses the entire query file to ascertain mentioned fields and their respective messages, after which it inspects the message file to calculate the offset of those fields inside the corresponding messages. Finally, it uses these offsets to generate eBPF code which extracts values from every calculated offset, and performs the corresponding operations in the order in which they are present in the query file.

4 EVALUATION

We now evaluate the benefits of using Pyramis in several ways. First, we show the expressiveness of Pyramis by using it to write specifications for several key components of the 5G packet core. Next, we evaluate the reference implementations auto-generated by the Pyramis translator and show that they perform at least as well as hand-optimized implementations.

5G Procedures	Throughput (req/sec)	
	Proprietary Implementation	Pyramis Implementation
Initial Registration	329	391
Session Establishment	116	130

Table 2: Performance of Pyramis generated code

4.1 Pyramis specifications

Using Pyramis, we write the specification of two components, Access and Mobility Function (AMF) and Session Management Function (SMF), the main components in the control plane of 5G packet core. We do not implement all the signaling messages of 5G, but focus on the initial registration procedure that is used by a mobile user to register and authenticate with the network, and the session establishment procedure that is used to set up a data session before data transfer through a mobile network. We write the code to handle all messages of the initial registration callflow at AMF, and all steps of the session establishment callflow at SMF using Pyramis. Note that the specification for a single step in the session establishment callflow that was spread across various documents as described in §1 was captured in a single place using Pyramis. The fact that we could implement these packet core components in Pyramis shows that Pyramis is expressive enough to describe complex multi-tier systems.

We now evaluate how writing specifications in Pyramis compares with writing code directly in a general purpose programming language (GPL), in terms of Lines of Code (LoC). For the AMF and SMF 5G core components described above, we consider the code for only the simplified initial registration and session establishment callflows from the proprietary production-grade 5G core implementation [1], and compare the LoC in this implementation with the LoC in the Pyramis DSL code written for these two callflows at AMF and SMF. Table 1 shows the LoC comparison between Pyramis DSL code and the conventional implementations written without Pyramis. Note that we omit common code like the message descriptions (message files), complex application layer logic (UDF files) which are present in both implementations from our calculations. We see from this table that the programmer has to write 2–3× more code without Pyramis for the 5G core components, which shows that the high-level abstractions of Pyramis can reduce developer effort when building complex systems. In addition to LoC savings, the code written in Pyramis is also more readable and has fewer implementation details, making it more accessible to domain experts without programming expertise.

4.2 Pyramis auto-generated code

Next, we use the Pyramis translator to auto-generate C++ code from the Pyramis specification of the AMF and SMF. We test our Pyramis generated components with other components of the packet core obtained from a proprietary production grade 5G core implementation [1], and we find that the AMF and SMF code auto-generated by Pyramis can inter-operate with the other 5G core components seamlessly. We also evaluate the performance of our auto-generated AMF and SMF, and compare it with the performance of the optimized production-grade AMF and SMF components. We use the

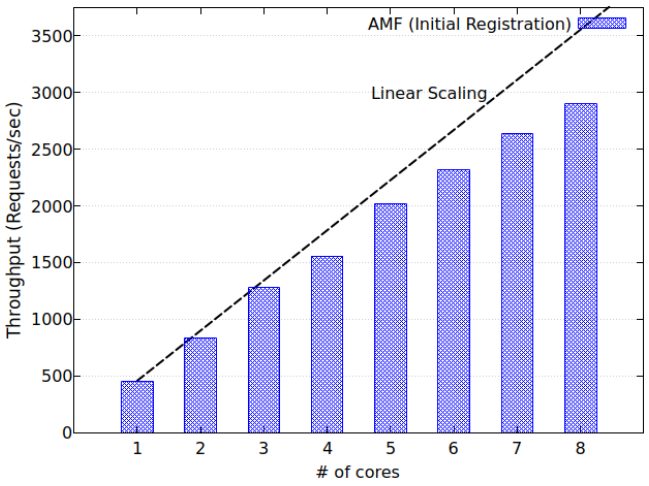
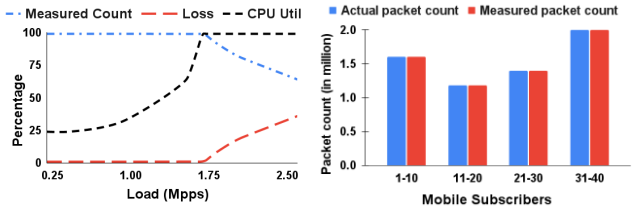


Figure 3: Multicore Scalability



(a) COUNT query **(b) GROUPBY query**

Figure 4: Measurement code correctness

load testing harness that was part of the production 5G core implementation. We test the AMF with a workload of initial registration requests and we test the SMF with a workload of session establishment requests. We measure the maximum throughput of the components in terms of number of registration or session setup requests handled per second. Table 2 shows the throughput of the Pyramis generated components and the production-grade components, both running on a single CPU under similar test conditions. We see from this table that the C++ code generated by the Pyramis translator is able to perform as well as a production-grade optimized implementation of AMF and SMF. Next, we repeat the load test on AMF by giving increasing number of cores to the AMF component. Note that the programmer does not have to do anything different in the Pyramis specification, and the translator automatically performs multi-threading for multicore scalability. Figure 3 shows the throughput for the Pyramis generated AMF. We see from the figure that Pyramis can generate multicore scalable components out-of-the-box without any additional programmer effort. These result show that building multi-tier systems with Pyramis does not hurt performance in any way. In fact, the Pyramis translator is an ideal place to plug in various performance optimizations and logic for multi-threading, so that they can be reused across multiple system implementations.

We now evaluate the correctness of eBPF measurement code generated using the Pyramis translator. We generate measurement code for two queries in the context of the 5G packet core. After a certain period of inactivity, the RAN may decide to release the wireless resources assigned to the idle UE, which triggers the AN release procedure in the packet core. The AN release procedure

marks the various data sessions of a UE as dormant and some state pertaining to the UE is released.

Query 1 uses the *FILTER* keyword to identify AN release packets belonging to the mobile subscriber of interest, and then simply counts the number of packets passing that filter using the *COUNT* keyword. Figure 4a shows the percentage of packets measured using Pyramis generated measurement code as opposed to the actual generated load. Pyramis generated measurement code is able to count every packet correctly until the CPU bottlenecks (utilization 100%), after which packets are dropped at the NIC. Query 2 uses the *GROUPBY* keyword to group AN release packets belonging to each mobile subscriber, and then uses the *COUNT* operation to count the number packets for every subscriber. Figure 4b compares the actual number of packets sent for a fixed number of mobile subscribers by the load generator to the number calculated by Pyramis generated measurement code. Figures 4a and 4b show the correctness of the generated measurement code as the count matches the exact number of packets of interest generated by the load generator.

5 RELATED WORK

Frameworks for modular network packet processing. Prior work has proposed several DSLs that allow for fast development of modular packet processing software by providing high-level constructs that abstract out the common tasks required in network packet processing. The Click [24] router has been a seminal work in this space. Click enables users to build a modular router using reusable elements for packet classification, queuing, scheduling, and other such functions. FastClick [11], ClickNF [20], ClickNP [25] extend the Click router to work in various different settings. Rubik [27] is a DSL written in Python which provides abstractions to build network protocol stacks for middleboxes. P4 [13] is a DSL to specify packet processing in a target-independent manner that can be compiled to run on a variety of targets from software switches to programmable hardware. mOS [23] is a framework over the mTCP network stack that allows the programmer to build stateful middleboxes by exposing high-level abstractions related to TCP level flow processing and allowing programmers to write event blocks for various TCP flow level events. All of these systems deal with specifying network packet processing at a single node in a multi-tier system and are complementary to our work that seeks to specify the interfaces, messages, and processing across multiple nodes in a multi-tier system.

L7 protocol parsers. Prior work also proposes high-level specifications for L7 (application layer) protocol messages, which can then be compiled into efficient packet parsers. Binpac [31] provides a declarative language along with a compiler to semantically analyse complex L7 network protocols. Protocol parsers built using Binpac were used in the Bro [32] network intrusion detection system. UltraPAC [28] builds on BinPAC to provide compiler optimizations. FlowSifter [29] introduces a new grammar model (Counting Regular Grammars) and a corresponding automata model (Counting Automata) to parse and extract fields from context sensitive application protocols. Work in this field also does protocol reverse engineering [14, 16, 34] to automatically infer application level protocol specifications. Our work is complementary to such efforts, and we can leverage such ideas to parse/encode/decode messages in our translator as well.

RPC frameworks. Popular RPC frameworks like Thrift [10] and gRPC [22] also provide abstractions to define and expose services. Both of these frameworks expect the programmer to define message formats, which are then compiled to a binary protocol during runtime facilitating encoding and decoding. These frameworks albeit powerful, only help in defining a message interface to another service running on another node in the multi-tier system, and generate code to serialize/deserialize messages received on the wire according to the defined interfaces, which as we have discussed before is only a part of the multi-tier system definition. Having said that, code generated by these frameworks to decode/encode messages into their respective transmission formats on the wire, could easily be used along with Pyramis using the *ENCODE* and *DECODE* keywords.

6 CONCLUSION

This paper describes Pyramis, a DSL to ease the specification and development of large multi-tier systems. Developers write the system specification using the high-level abstractions and language constructs provided by Pyramis. These specifications can then be used to auto-generate optimized C++ reference implementations of the various components by the Pyramis translator. The translator can also autogenerate other software that depends on the specifications, e.g., we generate eBPF-based performance measurement code from the message specifications and user-specified queries. Specifications written using Pyramis are more concise and easier for non-programmers to work with as compared to a reference implementation in a general purpose language, because the Pyramis translator automatically infers several implementation details without requiring them to be specified in the DSL code. Specifications in Pyramis are also precise and unambiguous by construction, unlike verbose text-based specifications that may be difficult to interpret and implement sometimes. We show that reference implementations auto-generated from Pyramis perform at least as well as hand-optimized implementations.

We envision a future where a specification language like Pyramis is adopted to write easy-to-understand and unambiguous specifications of large multi-tier systems like the mobile telecom systems. The standardization bodies like 3GPP will come up with Pyramis specifications to describe the system, and competing vendors will optimize the translator to differentiate their implementations from each other. Translators can also be developed to autogenerate other auxiliary software besides the performance measurement code we have shown, e.g., compliance checking code that is often used by telecom regulators to verify that the various telecom software components adhere to the specifications. Such software is developed at significant manual effort today, but can be easily auto-generated when specifications are written in a DSL. We plan to open-source the Pyramis language and translator and evangelize the adoption of this specification language in various 3GPP fora.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their comments, and Kapil Gokhale from the 5G Testbed project for providing invaluable insights on the Pyramis Domain Specific Language.

REFERENCES

[1] 2021. Proprietary 5G core implementation. Indigenuous 5G testbed project, IIT Bombay.

[2] 3GPP. 2022. *5G Security Assurance Specification (SCAS); User Plane Function (UPF); Stage 3.* Technical Specification (TS) 33.513. 3rd Generation Partnership Project. http://www.3gpp.org/DynaReport/33.513.htm Version 17.6.1.

[3] 3GPP. 2022. *5G System; Session Management Policy Control Service; Stage 3.* Technical Specification (TS) 29.512. 3rd Generation Partnership Project. http://www.3gpp.org/DynaReport/29512.htm Version 17.6.0.

[4] 3GPP. 2022. *5G System; Session Management Services; Stage 3.* Technical Specification (TS) 29.502. 3rd Generation Partnership Project. http://www.3gpp.org/DynaReport/29502.htm Version 17.4.0.

[5] 3GPP. 2022. *Access and Mobility Management Services; Stage 3.* Technical Specification (TS) 29.518. 3rd Generation Partnership Project. http://www.3gpp.org/DynaReport/29518.htm Version 17.6.1.

[6] 3GPP. 2022. *Interface between the Control Plane and the User Plane nodes.* Technical Specification (TS) 29.244. 3rd Generation Partnership Project. http://www.3gpp.org/DynaReport/29244.htm Version 17.4.0.

[7] 3GPP. 2022. *NG Access Protocol (NGAP) protocol for 5G System (5GS); Stage 3.* Technical Specification (TS) 38.413. 3rd Generation Partnership Project. http://www.3gpp.org/DynaReport/38413.htm Version 17.6.1.

[8] 3GPP. 2022. *Procedures for the 5G System (5GS).* Technical Specification (TS) 23.502. 3rd Generation Partnership Project. http://www.3gpp.org/DynaReport/23502.htm Version 17.4.0.

[9] 3GPP. 2022. *System architecture for the 5G System (5GS).* Technical Specification (TS) 23.501. 3rd Generation Partnership Project. http://www.3gpp.org/DynaReport/23501.htm Version 17.4.0.

[10] Apache. 2023. Apache Thrift. https://github.com/apache/thrift.

[11] Tom Barbette, Cyril Soldani, and Laurent Mathy. 2015. Fast userspace packet processing. In *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS).*

[12] Daniel Borkmann. 2021. tc-bpf(8) — Linux manual page.

[13] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* (2014).

[14] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. 2007. Polyglot: Automatic Extraction of Protocol Message Format Using Dynamic Binary Analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security.*

[15] Cilium. 2023. Cilium: eBPF-based Networking, Security, and Observability. https://github.com/cilium/cilium.

[16] Paolo Milani Comparetti, Gilbert Wondracek, Christopher Kruegel, and Engin Kirda. 2009. Prospex: Protocol Specification Extraction. In *2009 30th IEEE Symposium on Security and Privacy.*

[17] Coroot. 2023. Open-source observability augmented with actionable insights. https://github.com/coroot/coroot.

[18] Deepflowio. 2023. Application Observability using eBPF. https://github.com/deepflowio/deepflow.

[19] Matt Fleming. 2017. A thorough introduction to eBPF. https://lwn.net/Articles/740157/.

[20] Massimo Gallo and Rafael Laufer. 2018. Clicknf: a modular stack for custom network functions. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18).*

[21] gojue. 2023. Capture SSL/TLS text content without a CA certificate using eBPF. https://github.com/gojue/ecapture.

[22] Google. 2024. gRPC – An RPC library and framework. https://github.com/grpc/grpc.

[23] Muhammad Asim Jamshed, YoungGyoun Moon, Donghwi Kim, Dongsu Han, and KyoungSoo Park. 2017. mOS: A Reusable Networking Stack for Flow Monitoring Middleboxes. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17).*

[24] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. 2000. The Click Modular Router. *ACM Trans. Comput. Syst.* (2000).

[25] Bojie Li, Kun Tan, Layong Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. 2016. Clicknp: Highly flexible and high performance network processing with reconfigurable hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference.*

[26] Hao Li, Chengchen Hu, Junkai Hong, Xiyu Chen, and Yuming Jiang. 2015. Parsing application layer protocol with commodity hardware for SDN. In *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS).*

[27] Hao Li, Changhao Wu, Guangda Sun, Peng Zhang, Danfeng Shan, Tian Pan, and Chengchen Hu. 2021. Programming Network Stack for Middleboxes with Rubik.. In *NSDI.*

[28] Zhichun Li, Gao Xia, Hongyu Gao, Yi Tang, Yan Chen, Bin Liu, Junchen Jiang, and Yuezhou Lv. 2010. Netshield: massive semantics-based vulnerability signature matching for high-speed networks. *ACM SIGCOMM Computer Communication Review* (2010).

[29] Chad Meiners, Eric Norige, Alex X. Liu, and Eric Torng. 2012. FlowSifter: A counting automata approach to layer 7 field extraction for deep flow inspection. In *2012 Proceedings IEEE INFOCOM.*

[30] Netobserve. 2023. Network Observability eBPF Agent. https://github.com/netobserv/netobserv-ebpf-agent.

[31] Ruoming Pang, Vern Paxson, Robin Sommer, and Larry Peterson. 2006. Binpac: A Yacc for Writing Application Protocol Parsers. In *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement.*

[32] Vern Paxson. 1999. Bro: a system for detecting network intruders in real-time. *Computer Networks* (1999).

[33] Ben Pfaff, Keith Amidon, Justin Pettit, Martin Casado, Teemu Koponen, and Scott Shenker. 2009. Extending Networking into the Virtualization Layer. (2009).

[34] Yipeng Wang, Xiaochun Yun, M. Zubair Shafiq, Liyan Wang, Alex X. Liu, Zhibin Zhang, Danfeng Yao, Yongzheng Zhang, and Li Guo. 2012. A semantics aware approach to automated reverse engineering unknown protocols. In *2012 20th IEEE International Conference on Network Protocols (ICNP).*