ImpNet: Programming Software-Defied Networks Using Imperative Techniques*

Mohamed A. El-Zawawy[1,2,†]
[1]College of Computer and Information Sciences
Al Imam Mohammad Ibn Saud Islamic University
(IMSIU)
Riyadh
Kingdom of Saudi Arabia

[2]Department of Mathematics
Faculty of Science
Cairo University
Giza 12613
Egypt
maelzawawy@cu.edu.eg

Adel I. AlSalem
College of Computer and Information Sciences
Al Imam Mohammad Ibn Saud Islamic University
(IMSIU)
Riyadh
Kingdom of Saudi Arabia
alsalem@ccis.imamu.edu.sa

Abstract: Software and hardware components are basic parts of modern networks. However the software component is typical sealed and function-oriented. Therefore it is very difficult to modify these components. This badly affected networking innovations. Moreover, this resulted in network policies having complex interfaces that are not user-friendly and hence resulted in huge and complicated flow tables on physical switches of networks. This greatly degrades the network performance in many cases.
Software-Defined Networks (SDNs) is a modern architecture of networks to overcome issues mentioned above. The idea of SDN is to add to the network a controller device that manages all the other devices on the network including physical switches of the network. One of the main tasks of the managing process is *switch learning*; achieved via programming physical switches of the network by adding or removing rules for packet-processing to/from switches, more specifically to/from their flow tables.
A high-level imperative network programming language, called *ImpNet*, is presented in this paper. *ImpNet* enables writing efficient, yet simple, and powerful programs to run on the controller to control all other network devices including switches. *ImpNet* is compositional, simply-structured, expressive, and more importantly imperative. The syntax of *ImpNet* together two types of operational semantics to contracts of *ImpNet* are presented in the paper. The proposed semantics are of the static and dynamic types. Two modern application programmed using *ImpNet* are shown in the paper as well. The semantics of the applications are shown in the paper also.

Key–Words: Network programming languages, Controller-switch architecture, Static operational semantics, Dynamic operational semantics, Syntax, *ImpNet*, Software-defined networks.

1 Introduction

A network is a group of appliances connected to exchange data. Among these appliances are switches forwarding data depending on MAC addresses, routers forwarding data depending on IP addresses, and firewalls taking care of forbidden data. The network appliances are connected using a model that efficiently allows forwarding, storing, ignoring, tagging, and providing statistics about data moving in the network. Some of the network appliances, like routers [29, 24], are special in their functionality as they have some control over the network. This enables routers to compute and determine routes of data in the network. Of course different networks have different characteristics and abilities.

In 2011, the Open Networking Foundation [43], suggested removing the control owned by different network appliances and adding, instead, a general-purpose appliance, controller, to program different network appliances and querying data flowing in the network. The impact of this simple suggestion is huge; giant networks do not need special-purpose, complex, expensive switches any more. In such net-

*This is an extended and revised version of [22].
†Corresponding author.

works, cheap programmable switches can be used and programmed to configure and optimize networks via writing programs [28] running on controllers.

Software-Defined Networks (SDNs) [15] are networks established using the controller-switch architecture. A precise implementation of this architecture is OpenFlow [7] used to achieve various network-wide applications such as monitoring data flow, balancing switch load, network management, controlling appliances access, detection of service absence, host mobility, and forwarding data center. Therefore SDNs caused the appearance of network programming languages [26, 27, 25, 16].

This paper presents *ImpNet*, an imperative high-level network programming language. *ImpNet* expresses commands enabling controllers to program other network appliances including switches. *ImpNet* has a clear and simply-structured syntax based on classical concepts of imperative programming that allow building rich and robust network applications in a natural way. *ImpNet* can be realized as a generalization of Frenetic [32] which is a functional network programming language. This is clear by the fact that the core of programs written in *ImpNet* and Frenetic is based on a query result in the form of stream of values (packets, switches IDs, etc.). Commands for treating packets in *ImpNet* include constructing and installing (adding to flow tables of switches) switch rules. *ImpNet* supports building simple programs to express complex dynamic functionalities like load balancing and authentication. *ImpNet* programs can also analyze packets and historical traffic patterns.

The current paper presents both syntax and semantics of *ImpNet*. Actually two types of precise operational semantics for *ImpNet* are presented in this paper; static and semantics. Dynamic semantics[1] are very useful for studying and analyzing programs at run time. Such semantics have a wide range of applications in case of network programs. This is so as the network programs tend to be event-driven and hence their run times are much longer than that of many other application-programs. Therefore a formal definitions for run times behaviors (semantics) can easily be employed to achieve runtime verifications for controller programs. Moreover two *ImpNet* programs achieving two important controller applications are also presented in this paper together with their precise operational semantics.

The current paper is an extended and revised version of [22]. The current paper extends the work of [22] by supporting the theoretical foundations *ImpNet* via a dynamic operation semantics for its programs. The dynamic semantics is necessary for achieving many important dynamics verifications for network programs. The extensions are mainly included in Section 4 and Figures 8 and 9.

[1]Dynamic semantics can be realized as a perspective on programming languages semantics that models the increase of data in time.

Motivation

The motivation of this paper is the lack of a simple syntax for an imperative network programming language. Yet, a stronger motivation is that most existing network programming languages are not supported theoretically (using static operational semantics, dynamic operational semantics, type systems, program logics like Floyd–Hoare logic, etc.).

Contributions

Contributions of this paper are the following.

1. A new simply-structured syntax for an imperative network programming language; *ImpNet*.
2. A static operational semantics (in the form of states and inference rules) for constructs of *ImpNet*.
3. A dynamic operational semantics for constructs of *ImpNet*.
4. Two detailed examples of programs constructed in *ImpNet* with their precise operation semantics.

Organization

The rest of this paper is organized as following. Section 2 reviews related work. Section 3 presents the syntax and static operational semantics of *ImpNet*. The proposed static semantics are operational and hence consists of states and inference rules presented in Section 3. A dynamic operational semantics of *ImpNet* programs is introduced in Section 4. Two detailed examples of programmes built in *ImpNet* are presented in Section 5. This section also explains how the two examples can be assigned precise static operational semantics using our proposed static operational semantics. Section 6 gives directions for future work and Section 7 concludes the paper.

2 Related Work

This section presents work most related to that presented in the current paper. One of the early attempts to develop software-defined networking (SDN) is NOX [14] based on ideas from [13] and 4D [12]. On the switch-level, NOX uses explicit and callbacks rules for packet-processing. Examples of applications

$$\begin{array}{rcl}
 & & x \in \text{lVar} \qquad Q \in \text{Queries} \qquad n \in \text{Integers} \\
et \in \text{Eventrans} & ::= & n \mid \text{Lift}(x, \lambda t.f(t)) \mid \text{ApplyLft}(x, \lambda t.f(t)) \mid \text{ApplyRit}(x, \lambda t.f(t)) \mid \\
 & & \text{Merge}(x_1, x_2) \mid \text{MixFst}(A, x_2, x_3) \mid \text{MixSnd}(A, x_2, x_3) \mid \\
 & & \text{Filter}(x, \lambda.f(t)) \mid \text{Once}(x) \mid \text{MakForwRule}(x) \mid \text{MakeRule}(x) \\
S \in \text{Stmts} & ::= & x := et \mid S_1; S_2 \mid \text{AddRules}(x) \mid \text{Register} \mid \text{Send}(x) \\
 & & \text{If } (x) \text{ then } S_1 \text{ else } S_2 \mid \text{While } (x) \text{ do } S \\
D \in \text{Defs} & ::= & \epsilon \mid x := Q \mid DD. \\
p \in \text{Progs} & ::= & D \gg S.
\end{array}$$

Figure 1: ImpNet Syntax.

that benefitted from NOX are load balancer [11] and the work in [9, 10]. Many directions for improving platforms of programming networks include Maestro [7] and Onix [8], which uses distribution and parallelization to provide better performance and scalability.

A famous programming language for networks is Frenetic [32, 33] which has two main components. The first component is a collection of operators that are source-level. The operators aim at establishing and treating streams of network traffic. These operators also are built on concepts of functional programming (FP) and query languages of declarative database. Moreover the operators support a modular design, a cost control, a race-free semantics, a single-tier programming, and a declarative design. The second component of Frenetic is a run-time system. This system facilitates all of the actions of adding and removing low-level rules to and from flow tables of switches. One advantage of *ImpNet*, the language presented in this paper, over Frenetic is that *ImpNet* is imperative. Therefore *ImpNet* paves the way to the appearance of other types of network programming languages such as object-oriented network programming langues and context-oriented network programming languages.

Other examples to program network components though high-level languages are NDLog and NetCore [6]. NetCore provides an integrated view of the whole network. NDLog is designed in an explicitly distributed fashion.

As an extension of Datalog, NDLog [30, 31] was presented to determine and code protocols of routing [29], overlay networks, and concepts like hash tables of distributed systems. *ImpNet* (presented in this paper), Frenetic, and NDLog can be classified as high-level network programming languages. While NDLog main focus is overlay networks and routing protocols, Frenetic (in a functional way) and *ImpNet* (in an imperative way) focus on implementing packet processing such as modifying header fields. Therefore *ImpNet* equips a network programmer with a modular view of the network which is not provided by NDLog and Frenetic. This is supported by the fact that a program in NDLog is a single query that is calculated on each router of the network. One advatnage of network programming langauges (*ImpNet*) is saving routing energy [41].

Energy Efficient Routing with Transmission Power Control based Biobjective Path Selection Model for Mobile Ad-hoc Network

The switch component [39] of networks can be programmed via many interfaces such as OpenFlow platform. Examples of other platforms include Shangri-La [40] and FPL-3E [42], RouteBricks [37], Click modular router [34], Snortran [35] and Bro [36]. The idea in Shangri-La [40] and FPL-3E [42] is to produce certain hardware for packet-processing from high-level programs that achieves packet-processing. In RouteBricks [37], stock machines are used to improve performance of program switches. As a modular approach, the platform of Click modular router [34], enables programming network components. This system focuses on software switches in the form of Linux kernel code. For the sake of intrusions detection and preserving network security, Snortran [35] and Bro [36] enable coding monitoring strategies and robust packet-filtering. One advantage of *ImpNet*, the language presented in this paper, over all the related work is that *ImpNet* overcomes the disadvantage of most similar languages of focusing on controlling a single device.

There are many possible network applications for dynamics semantics of *ImpNet*. The K-random search in peer-to-peer networks using dynamic semantic data replication [1] can be better verified using dynamic semantics of the nature proposed in this paper. The correctness of dynamic information retrieval in P2P networks is achieved using dynamics semantics platform [2]. Dynamic reconfiguration of networked services can be carried out using dynamic semantic in the shape of interoperability framework [3] like that of this paper. In an attractor network, spreading activation with latching dynamics can be realized using automatic dynamic semantic similar to the one proposed in this paper [4].

3 Syntax and Static Operational Semantics

This section presents the syntax and static operational semantics of *ImpNet*, a high-level programming language for SDN networks using the switch-controller architecture. Figure 1 shows the syntax of *ImpNet*. Figures 2 and 3 present the static operational semantics of *ImpNet* constructs. The proposed semantics is operational and its states are defined in the following definition.

Definition 1
1. *$t \in$ Types $= \{$int, Switch IDs, Packet, (Switch IDs, int, bool)$\} \cup \{(t_1, t_2) \mid t_1, t_2 \in$ Types$\}$.*
2. *$v \in$ Values $=$ Natural numbers $\cup$ Switch IDs $\cup$ Packets $\cup$ Switch IDs $\times$ Natural numbers $\times$ Boolean values $\cup \{(v_1, v_2) \mid v_1, v_2 \in$ Values$\}$. The expression $v : t$ denotes that the type of the value v is t.*
3. *$ev \in$ Events $= \{(v_1, v_2, \ldots, v_n) \mid \exists t(\forall i\ v_i : t)\}$.*
4. *Actions$= \{$sendcontroller, sendall, sendout, change(h,v) $\}$.*
5. *$r \in$ Rules $=$ Patterns $\times$ Acts.*
6. *$rl \in$ RlLst $= \{[r_1, r_2, \ldots, r_n] \mid r_i \in$ Rules$\}$.*
7. *$ir \in$ Intial-rule-assignment $=$ SwchIds $\times$ Rules.*
8. *$\sigma \in$ SwchSts $=$ Flow-tables $=$ SwchIds $\to$ RlLst.*
9. *$\gamma \in$ VarSts $=$ Var $\to$ Events $\cup$ RlLst.*
10. *$s \in$ States $=$ SwchSts $\times$ VarSts $\times$ RlLst.*

A program in *ImpNet* is a sequence of queries followed by a statement. The result of each query is an event which is a finite sequence of values. The event concept is also used in Frenetic. However an event in Frenetic is an infinite sequence of values. A value is an integer, a switch ID, a packet, a triple of a switch ID, an integer, and a Boolean value, or a pair of two values. Each value has a type of the set *Types*. In this paper, we focus on the details of statements as this is the most interesting part in a network programming language.

The query part of a network language is there to enable reading the status of the network. Typically, queries contain commands for

- dividing packets via grouping according to values of header fields,
- splitting packets according to arrival time or values of header fields,
- filtering packets in the network according to a given pattern,
- minimizing the volume of returned values, and
- summarizing results using size or number of packets.

Possible actions taken by a certain switch on a certain packet are *sendcontroller*, *sendall*, *sendout*, or *change(h,v)*. The action *sendcontroller* sends a packet to the controller to process it. The action *sendall* sends the packet to all other switches. The action *sendout* sends the packet out of the switch through a certain port. The action *change(h,v)* modifies the header field h of the packet to the new value v.

A rule in our static operational semantics is a pair of *pattern* and *action* where *pattern* is a form that concretely describes a set of packets and *action* is the action to be taken on elements of this set of packets. Rules are stored in tables (called *flow tables*) of switches. *Intial-rule-assignment* represents an initial assignment of rules to flow tables of switches.

A state in the proposed static operational semantics is a triple (σ, γ, ir). In this triple γ captures the current state of the program variables and hence is a map from the set of variables to the set of events and rule lists. This is so because in *ImpNet* variables may contain events or rule lists. The symbol σ captures the current state of flow tables of switches and hence is a map from *Switche IDs* to rule lists. Finally, ir is an initial assignment of rules assigned to switches but have not been registered yet (have not been added to γ yet).

There are five type of statements in *ImpNet*. The assignment statement $x := ef$ assigns the result of an event transformer (et) to the variable x. The statement *AddRules(x)* adds the switch rules stored in x to the reservoir of initially assigned rules. These are rules that are assigned to switches but are not added to flow tables yet. The statement *Register* makes the initial assignments permeant by adding them to flow tables of switches. The statement *Send(x)* sends specific packets to be treated in a certain way at certain switches. To keep a record of of actions taken on

$$\frac{v_i : t \qquad \gamma(x) = (v_1, v_2, \ldots, v_n)}{\text{Lift}(x, \lambda t.f(t)) : \gamma \to (f(v_1), f(v_2), \ldots, f(v_n))}\ (\text{Lift}^s)$$

$$\frac{\gamma(x_1) = (v_1, v_2, \ldots, v_n) \qquad \gamma(x_2) = (w_1, w_2, \ldots, w_n)}{\text{Merge}(x_1, x_2) : \gamma \to ((v_1, w_1), (v_2, w_2), \ldots, (v_n, w_n))}\ (\text{Merge}^s)$$

$$\frac{\gamma(x) = (v_1, v_2, \ldots, v_n) \qquad A = \{i \mid f(v_i) = \text{true}\}}{\text{Filter}(x, \lambda.f(t)) : \gamma \to (\ldots, v_i, \ldots \mid i \in A)}\ (\text{Filter}^s)$$

$$\frac{v_i : t \qquad \gamma(x) = ((v_1, v_1'), (v_2, v_2'), \ldots, (v_n, v_n'))}{\text{ApplyLft}(x, \lambda t.f(t)) : \gamma \to ((f(v_1), v_1'), (f(v_2), v_2'), \ldots, (f(v_n), v_n'))}\ (\text{App}_1^s)$$

$$\frac{v_i' : t \qquad \gamma(x) = ((v_1, v_1'), (v_2, v_2'), \ldots, (v_n, v_n'))}{\text{ApplyRit}(x, \lambda t.f(t)) : \gamma \to ((v_1, f(v_1')), (v_2, f(v_2')), \ldots, (v_n, f(v_n')))}\ (\text{App}_2^s)$$

$$\frac{\text{type}(x) \in \text{Types}}{\text{Once}(x) : \gamma \to (\underbrace{x, x, \ldots, x}_{n\_times})}\ (\text{Once}^s)$$

$$\frac{\gamma(x_1) = (v_1^1, v_2^1, \ldots, v_n^1) \quad \gamma(x_2) = (v_1^2, v_2^2, \ldots, v_n^2) \quad A_1 = A \cup \{v_1^1\} \quad \forall i > 1. A_i = A_{i-1} \cup \{v_i^1\}}{\text{MixFst}(A, x_1, x_2) : \gamma \to ((A_1, v_1^2), (A_2, v_2^2), \ldots, (A_n, v_n^2))}\ (\text{Mix}_1^s)$$

$$\frac{\gamma(x_1) = (v_1^1, v_2^1, \ldots, v_n^1) \quad \gamma(x_2) = (v_1^2, v_2^2, \ldots, v_n^2) \quad A_1 = A \cup \{v_1^2\} \quad \forall i > 1. A_i = A_{i-1} \cup \{v_i^2\}}{\text{MixSnd}(A, x_1, x_2) : \gamma \to ((v_1^1, A_1), (v_2^1, A_2), \ldots, (v_n^1, A_n))}\ (\text{Mix}_2^s)$$

$$\frac{\gamma(x) = ((v_1^1, v_1^2, v_1^3), (v_2^1, v_2^2, v_2^3), \ldots, (v_n^1, v_n^2, v_n^3))}{\text{MakForwRule}(x) : \gamma \to [(v_1^1, (v_1^3, sendout(v_1^2))), (v_2^1, (v_2^3, sendout(v_2^2))), \ldots, (v_n^1, (v_n^3, sendout(v_n^2)))]}\ (\text{MFR}^s)$$

$$\frac{\gamma(x) = ((v_1^1, a_1, v_1^2), (v_2^1, a_2, v_2^2), \ldots, (v_n^1, a_n, v_n^2))}{\text{MakeRule}(x) : \gamma \to [(v_1^1, a_1(v_2^1)), (v_1^2, a_2(v_2^2)), \ldots, (v_n^i, a_n(v_2^n))]}\ (\text{MkRl}^s)$$

Figure 2: Operational Semantics for Event Functions of ImpNet

$$\frac{et : \gamma \to u}{x := et : (\sigma, \gamma, ir) \to (\sigma, \gamma[x \mapsto u], ir)}\ (\text{Assgn}^s)$$

$$\frac{S_1 : (\sigma, \gamma, ir) \to (\sigma'', \gamma'', ir'') \qquad S_2 : (\sigma'', \gamma'', ir'') \to (\sigma', \gamma', ir')}{S_1; S_2 : (\sigma, \gamma, ir) \to (\sigma', \gamma', ir')}\ (\text{seq}^s)$$

$$\frac{\gamma(x) \in \text{Intial-rule-assignment}}{\text{AddRules}(x) : (\sigma, \gamma, ir) \to (\sigma, \gamma, ir \cup \gamma(x))}\ (\text{Addrl}^s)$$

$$\frac{}{\text{Register} : (\sigma, \gamma, ir) \to (\sigma \cup ir, \gamma, \emptyset)}\ (\text{Reg}^s)$$

$$\frac{\gamma(x) = ((v_1^1, v_1^2, v_1^3), (v_2^1, v_2^2, v_2^3), \ldots, (v_n^1, v_n^2, v_n^3)) \qquad \forall i.(v_2^i, v_3^i) \in \text{history}(v_1^i)}{\text{Send}(x) : (\sigma, \gamma, ir) \to (\sigma, \gamma, ir)}\ (\text{Send}^s)$$

Figure 3: Operational Semantics for Statements of ImpNet

Figure 4: Static Operational Semantics for ImpNet.

packets on different switches we assume a map called *history* from the set of switche IDs to the set of lists of pairs of packets and taken actions. This map is used in the Rule (Send^s). Static operational semantics of these statements are given in Figure 3. Judgments of inference rules in this figure have the form $S : (\sigma, \gamma, ir) \to (\sigma', \gamma', ir')$. This judgement reads as following. If the execution of S in the state (σ, γ, ir) ends then the execution reaches the state (σ', γ', ir').

Inference rules in Figure 3 use that in Figure 2 to get the semantics of the other important construct of *ImpNet* which is event transformers (et). Judgements of Figure 2 have the form $et : \gamma \to u$ meaning that the semantics of the transformer et in the variable state γ is u. The event transformer $\text{Lift}(x, \lambda t.f(t))$ applies the map $\lambda t.f(t)$ to values of the event in x (Rule (Lift^s)). The event transformer $\text{Filter}(x, \lambda.f(t))$ filters the event in x using the map $\lambda t.f(t)$ (Rule (Filter^s)). From a given set of actions A and two events x_1 and x_2 the event transformers $\text{MixFst}(A, x_1, x_2)$ and $\text{MixSnd}(A, x_1, x_2)$ create lists of rules (Rules (Mix_1^s) and (Mix_2^s)).

4 Dynamic Semantics

This section presents a dynamic semantics for constructs of *ImpNet* using Rewriting Logic Semantics [5]. An important concept to present the semantics is that of rewrite theories $(\prod_{ImpNet}, E_{ImpNet}, R_{ImpNet})$. This concept uses an extended version of the langauge syntax $\prod_{ImpNet}$, a group of equations, E_{ImpNet}, built on $\prod_{ImpNet}$, and set of rules, R_{ImpNet}, for $\prod_{ImpNet}$ constructs. Via a set of rearrangements, E_{ImpNet} is necessary to prepare the environment for applying the rules and hence E_{ImpNet} expresses no computational semantics. However R_{ImpNet} is the semantics component modeling in an irreversible way the computations. All in all, the semantics uses rewrite theories to define *ImpNet*. This semantics is built on equational logic [44] and hence it is allowed for terms to replace equal terms in any context. This adds to the power of the semantics. The idea is that equations are meant to be applied until arriving at a term matching the l.h.s. of one of the rules. This rule can then be used to do an irreversible transformation to the term. This type of semantics is typically efficiently executable.

The proposed semantics uses a modular framework named C. In this framework rules use only necessary configuration items. Hence configuration changes, e.g. adding stacks, do not imply modifying existing rules. Sequences, maps, and bags are necessary concepts to definitions of the C language. This paper uses the classical interception of these concepts as data-structures of the equational type. Sequences, maps, and bags are denoted by $\text{Sq}_e^{-\hookrightarrow-}$, $\text{Mp}_e^{-\hookrightarrow-}$, and $\text{Bg}_e^{-\hookrightarrow-}$, respectively, where $\hookrightarrow$ is a binary operator and e is the unit element. Therefore in our proposed dynamic semantics an environment (a state) is a finite bag of pairs. The domain of a function can easily be expressed as a bag of items.

Definition 2 gives a formal presentation of states (configurations) of the proposed dynamic semantics.

Definition 2
- $C^s = C \mid Sq_{.}^{-\hookrightarrow-}[C]$.
- $\sigma \in SwchSts = Mp_{.}^{-,-}[SwchIds, RlLst]$.
- $h \in History = Mp_{.}^{-,-}[SwchIds, [Packets, Acts]]$.
- $\gamma \in VarSts = Mp_{.}^{-,-}[Var, Events \cup RlLst]$.
- $confitm \in Configurations\ items = < C >_{C^s} \mid < \sigma >_{SwchSts} \mid < \gamma >_{VarSts} \mid < rl >_{RlLst} \mid < h >_{History}$.
- $conf \in Configurations = < Bg_{.}^{-}[confitm] >$.

The abstract syntax of *ImpNet*, history maps, maps, sequences, and bags are used as configuration constructors. The configuration of *ImpNet* include five components:

- $< C >_{C^s}$ including the computations,
- $< \sigma >_{\text{SwchSts}}$ capturing flow tables of the physical switch of the concerned network,
- $< \gamma >_{\text{VarSts}}$ holding the mapping for the program variables including event variables,
- $< rl >_{\text{RlLst}}$ including the list of rules to be registered, and finally
- $< h >_{\text{History}}$ capturing the history of packets treatment of the network.

Algebraic structures in definitions of configurations achieve the context-sensitivity in C definitions. The sequentialization operator $\hookrightarrow$ in the dynamic semantics also contributes to the context-sensitivity.

Figure 5 presents the complete C definitions of *ImpNet*. Typically C definitions include a single syntactic category C. This category is intended to be a minimal infrastructure for terms definitions; not to be a parsing or type-checking tool. Typically in such semantics, sorts are equivalent to syntactic categories and operations are equivalent to productions. Hence algebraic signatures somehow coincide with context-free notations. In the model C, Boolean values are treated using special integer values.

$$
\begin{array}{rcl}
 & & x \in \text{lVar} \qquad Q \in \text{Queries} \qquad n \in \text{Integers} \\
C \in \text{Core} & ::= & x \mid n \mid C_1 \text{ op } C_2 \mid [] \mid Q \mid C - \lambda t.f(t) \mid (C, \lambda t.f(t)) \mid \\
 & & (\lambda t.f(t), C) \mid (C_1, C_2) \mid (A, C_1, C_2) \mid (C_1, A, C_2) \mid \\
 & & (C, f, \lambda.f(t)) \mid O(C) \mid F(C) \mid M(C) \mid C_1 := C_2 \mid \\
 & & C; \mid A(C) \mid R \mid S(C) \mid C_1 C_2 \text{If } (C_1)\ C_2 \mid \\
 & & \text{If } (C_1)\ C_2\ C_3 \mid \text{While } (C_1)\ C_2 \mid C_1 \gg C_2
\end{array}
$$

Figure 5: Core of ImpNet in C.

$C - \lambda t.f(t) = \text{Lift}(C, \lambda t.f(t))$	$(C, \lambda t.f(t)) = \text{ApplyLft}(C, \lambda t.f(t))$
$(\lambda t.f(t), C) = \text{ApplyRit}(C, \lambda t.f(t))$	$(C_1, C_2) = \text{Merge}(C_1, C_2)$
$(A, C_1, C_2) = \text{MixFst}(A, C_1, C_2)$	$(C_1, A, C_2) = \text{MixSnd}(A, C_1, C_2)$
$(C, f, \lambda.f(t)) = \text{Filter}(C, \lambda.f(t))$	$O(C) = \text{Once}(C)$
$F(C) = \text{MakForwRule}(C)$	$M(C) = \text{MakeRule}(C)$
$A(C) = \text{AddRules}(C)$	$R(C) = \text{Register}(C)$
$S(C) = \text{Send}(C)$	$\text{If } (C_1)\ C_2 = \text{If } (C_1) \text{ then } C_2$
$\text{If } (C_1)\ C_2\ C_3 = \text{If } (C_1) \text{ then } C_2 \text{ else } C_3$	$\text{While } (C_1)\ C_2 = \text{While } (C_1) \text{ do } C_2$

Figure 6: Desugaring of Core Constructs of Figure 5.

$C_1 \text{ op } C_2 = (C_1 \hookrightarrow \diamond \text{ op } C_2)$	$n \text{ op } C_2 = (C_2 \hookrightarrow n \text{ op } \diamond)$
$(C_1 := C_2) = (C_2 \hookrightarrow C_1 := \diamond)$	$C; = (C \hookrightarrow \diamond;)$
$\text{If } (C_1)\ C_2\ C_3 = (C_1 \hookrightarrow \text{If } (\diamond)\ C_2\ C_3)$	

Figure 7: Computational Structural Equations of Dynamic Semantics.

Figure 8: Dynamic Operational Semantics for ImpNet.

The concepts of well-structured and well-evaluated computations are necessary for recognizing start and final configurations of the dynamic semantics. Using equational reasoning of *ImpNet*s dynamic semantics of this section, an *ImpNet* computation C is *well-structured* if it amounts to a well-structured event transformers or list of statements in *ImpNet*. A computation is *well-evaluated* if it amounts to a value $v \in$ *Values* or to the unit computation ".".

Figures 8 and 9 present all components of dynamic semantics of *ImpNet*. Mostly, elements of these figures are self-describing. Wherever possible, it is preferred to desugar constructs of our derived language.

There are some special configurations of the dynamic semantics presented in this section. For a well-structures computation C, the configuration $<< C >_{C^s}, < \cdot >_{\text{SwchSts}}, < \cdot >_{\text{VarSts}}, < \cdot >_{\text{RlLst}}, < \cdot >_{\text{History}} >$ is the *start configuration*. For a "," or a value computation C, the configuration $<< C >_{C^s}, < \sigma >_{\text{SwchSts}}, < \gamma >_{\text{VarSts}}, < rl >_{\text{RlLst}}, < h >_{\text{History}} >$ is a *final configuration*.

The expression $C_1 \hookrightarrow C_2$ denotes processing C_1 before processing C_2. Therefore C_2 is somehow a frozen (from development) computation until it is its turn. The technique of the evaluation of the langue *ImpNet* is meant to be captured by the computation equations. For example, the conditional statement schedules calculating the condition first while the branches are frozen. The most intricate rules of Figure 9 are that of event transformers. This is so as most of these rules assume constrains on muliple event variables.

5 Controller Programs

This section presents several examples of programs constructed using the syntax of *ImpNet* (Figure 1). The first example constructs rules based on information stored in the variable x and then installs the established rules to flow tables of switches stored in z. This program has the following statements.

$$y = \text{MakeRule}(x);$$

$$z = \text{Lift}(z, \lambda t.(t, y));$$

$$\text{AddRules}(z);$$

$$Register;$$

The first statement of the program makes a rule for each value of the event stored in x. Then the second statement assigns these rules to switch IDs in the

$[] = .$
$C_1 C_2 = C_1 \hookrightarrow C_2$
C_1 op $C_2 \Longrightarrow C_1$ $\text{op}_{\text{int}} C_2$
If (C_1) C_2 $C_3 \Longrightarrow C_2$, where $C_1 \neq 0$
If (C_1) C_2 $C_3 \Longrightarrow C_3$, where $C_1 = 0$
Left:
$< x - \lambda t.f(t) \hookrightarrow C >_{C^s} < x \mapsto (v_1, v_2, \ldots, v_n), \gamma >_{\text{VarSts}} \Longrightarrow$
$< (f(v_1), f(v_2), \ldots, f(v_n)) \hookrightarrow C >_{C^s} < x \mapsto (v_1, v_2, \ldots, v_n), \gamma >_{\text{VarSts}}$
ApplyLft:
$< (x, \lambda t.f(t)) \hookrightarrow C >_{C^s} < x \mapsto ((v_1, w_1), (v_2, w_2), \ldots, (v_n, w_n)), \gamma >_{\text{VarSts}} \Longrightarrow$
$< ((f(v_1), w_1), (f(v_2), w_2), \ldots, (f(v_n), w_n)) \hookrightarrow C >_{C^s}$
$< x_1 \mapsto (v_1, v_2, \ldots, v_n), x_2 \mapsto (w_1, w_2, \ldots, w_n), \gamma >_{\text{VarSts}}$
ApplyRit:
$< (\lambda t.f(t), x) \hookrightarrow C >_{C^s} < x \mapsto ((v_1, w_1), (v_2, w_2), \ldots, (v_n, w_n)), \gamma >_{\text{VarSts}} \Longrightarrow$
$< ((v_1, f(w_1)), (v_2, f(w_2)), \ldots, (v_n, f(w_n))) \hookrightarrow C >_{C^s}$
$< x_1 \mapsto (v_1, v_2, \ldots, v_n), x_2 \mapsto (w_1, w_2, \ldots, w_n), \gamma >_{\text{VarSts}}$
Merge:
$< (x_1, x_2) \hookrightarrow C >_{C^s} < x_1 \mapsto (v_1, v_2, \ldots, v_n), x_2 \mapsto (w_1, w_2, \ldots, w_n), \gamma >_{\text{VarSts}} \Longrightarrow$
$< ((v_1, w_1), (v_2, w_2), \ldots, (v_n, w_n)) \hookrightarrow C >_{C^s}$
$< x_1 \mapsto (v_1, v_2, \ldots, v_n), x_2 \mapsto (w_1, w_2, \ldots, w_n), \gamma >_{\text{VarSts}}$
MixFst:
$< (A, x_1, x_2) \hookrightarrow C >_{C^s} < x_1 \mapsto (v_1, v_2, \ldots, v_n), x_2 \mapsto (w_1, w_2, \ldots, w_n), \gamma >_{\text{VarSts}} \Longrightarrow$
$< ((A_1, w_1), \ldots, (A_n, w_n)) = ((A \cup \{v_1\}, w_1), (A_1 \cup \{v_2\}, w_2), \ldots, (A_{n-1} \cup \{v_n\}, w_n))$
$\hookrightarrow C >_{C^s} < x_1 \mapsto (v_1, v_2, \ldots, v_n), x_2 \mapsto (w_1, w_2, \ldots, w_n), \gamma >_{\text{VarSts}}$
MixSnd:
$< (x_1, A, x_2) \hookrightarrow C >_{C^s} < x_1 \mapsto (v_1, v_2, \ldots, v_n), x_2 \mapsto (w_1, w_2, \ldots, w_n), \gamma >_{\text{VarSts}} \Longrightarrow$
$< ((v_1, A_1), \ldots, (v_n, A_n)) = ((v_1, A \cup \{w_1\}), (v_2, A_1 \cup \{w_2\}), \ldots, (v_n, A_{n-1} \cup \{w_n\}))$
$\hookrightarrow C >_{C^s} < x_1 \mapsto (v_1, v_2, \ldots, v_n), x_2 \mapsto (w_1, w_2, \ldots, w_n), \gamma >_{\text{VarSts}}$
Filter:
$< (C, f, \lambda.f(t)) \hookrightarrow C >_{C^s} < x \mapsto (v_1, v_2, \ldots, v_n), \gamma >_{\text{VarSts}} \Longrightarrow$
$< (\ldots, v_i, \ldots \mid f(v_i) = \text{ture}) \hookrightarrow C >_{C^s} < x \mapsto (v_1, v_2, \ldots, v_n), \gamma >_{\text{VarSts}}$
Once:
$< O(x) \hookrightarrow C >_{C^s} < x \mapsto v, \gamma >_{\text{VarSts}} \Longrightarrow < (v^1, \ldots, v^n) \hookrightarrow C >_{C^s} < x \mapsto v, \gamma >_{\text{VarSts}}$
MakForwRule:
$< F(C) \hookrightarrow C >_{C^s} < x \mapsto ((v_1^1, v_1^2, v_1^3), (v_2^1, v_2^2, v_2^3), \ldots, (v_n^1, v_n^2, v_n^3)), \gamma >_{\text{VarSts}} \Longrightarrow$
$< [(v_1^1, (v_1^3, sendout(v_1^2))), (v_2^1, (v_2^3, sendout(v_2^2))), \ldots, (v_n^1, (v_n^3, sendout(v_n^2)))]$
$\hookrightarrow C >_{C^s} < x \mapsto ((v_1^1, v_1^2, v_1^3), (v_2^1, v_2^2, v_2^3), \ldots, (v_n^1, v_n^2, v_n^3)), \gamma >_{\text{VarSts}}$
MakeRule:
$< M(C) \hookrightarrow C >_{C^s} < x \mapsto ((v_1^1, a_1, v_1^2), (v_2^1, a_2, v_2^2), \ldots, (v_n^1, a_n, v_n^2)), \gamma >_{\text{VarSts}} \Longrightarrow$
$< [[(v_1^1, a_1(v_2^1)), (v_1^2, a_2(v_2^2)), \ldots, (v_n^i, a_n(v_2^n))]]$
$\hookrightarrow C >_{C^s} < x \mapsto ((v_1^1, v_1^2, v_1^3), (v_2^1, v_2^2, v_2^3), \ldots, (v_n^1, v_n^2, v_n^3)), \gamma >_{\text{VarSts}}$
Assignment:
$< x := v \hookrightarrow C >_{C^s} < \gamma >_{\text{VarSts}} \Longrightarrow < C >_{C^s} < \gamma[x \mapsto v] >_{\text{VarSts}}$
AddRules:
$< A(x) \hookrightarrow C >_{C^s} < x \mapsto [rl, \ldots, rl_n], \gamma >_{\text{VarSts}} < rl >_{\text{RlLst}} \Longrightarrow$
$< C >_{C^s} < x \mapsto [rl, \ldots, rl_n], \gamma >_{\text{VarSts}} < [rl, \ldots, rl_n] \cup rl >_{\text{RlLst}}$
Register:
$< R \hookrightarrow C >_{C^s} < rl >_{\text{RlLst}} < \sigma >_{\text{SwchSts}} \Longrightarrow < C >_{C^s} < [] >_{\text{RlLst}} < rl \cup \sigma >_{\text{SwchSts}}$
Send:
$< S(C) \hookrightarrow C >_{C^s} < x \mapsto ((v_1^1, v_1^2, v_1^3), (v_2^1, v_2^2, v_2^3), \ldots, (v_n^1, v_n^2, v_n^3)), \gamma >_{\text{VarSts}} < h >_{\text{History}}$
$\Longrightarrow < C >_{C^s} < x \mapsto ((v_1^1, v_1^2, v_1^3), (v_2^1, v_2^2, v_2^3), \ldots, (v_n^1, v_n^2, v_n^3)), \gamma >_{\text{VarSts}}$
$< v_1^i \mapsto \{(v_2^i, v_3^i), h >_{\text{History}}$
$<$ While (C_1) $C_2 \hookrightarrow C >_{C^s} \Longrightarrow <$ If (C_1) $[C_2$While (C_1) $C_2 \hookrightarrow C] >_{C^s}$

Figure 9: Equations and Rules of Dynamic Semantics.

$(\emptyset, \{z \mapsto \{id_1, id_2\}, x \mapsto \{((srcport(80), sendall, \_), (inport(1), sendcontroller, \_))\}, [])$

$y = \textbf{MakeRule}(x);$

$(\emptyset, \{z \mapsto \{id_1, id_2\}, x \mapsto \{((srcport(80), sendall, \_), (inport(1), sendcontroller, \_))\},$

$y \mapsto \{(srcport(80), [sendall]), (inport(1), [sendcontroller])\}\}, \emptyset)$

$z = \textbf{Lift}(z, \lambda t.(t, y));$

$(\emptyset, \{z \mapsto \{(id_1, \gamma(y)), (id_2, \gamma(y))\},$

$x \mapsto \{((srcport(80), sendall, \_), (inport(1), sendcontroller, \_))\},$

$y \mapsto \{(srcport(80), [sendall]), (inport(1), [sendcontroller])\}\}, \emptyset)$

$\textbf{AddRules}(z);$

$(\emptyset, \{z \mapsto \{(id_1, \gamma(y)), (id_2, \gamma(y))\},$

$x \mapsto \{((srcport(80), sendall, \_), (inport(1), sendcontroller, \_))\},$

$y \mapsto \{(srcport(80), [sendall]), (inport(1), [sendcontroller])\}\}, \{(id_1, \gamma(y)), (id_2, \gamma(y))\})$

Register;

$(\{(id_1, \gamma(y)), (id_2, \gamma(y))\}, \{z \mapsto \{(id_1, \gamma(y)), (id_2, \gamma(y))\},$

$x \mapsto \{((srcport(80), sendall, \_), (inport(1), sendcontroller, \_))\},$

$y \mapsto \{(srcport(80), [sendall]), (inport(1), [sendcontroller])\}\}, \emptyset)$

Figure 10: Program 1.

$(\emptyset, \{z \mapsto \{id_1, id_2\}\}, [])$

$y = \textbf{SourceIps};$

$(\emptyset, \{z \mapsto \{id_1, id_2\},$

$y \mapsto \{(ip_1, pk_1), (ip_2, pk_2)\}\}, \emptyset)$

$y = \textbf{ApplyLft}(y, \lambda t.(t, \textbf{port}(t)));$

$(\emptyset, \{z \mapsto \{id_1, id_2\},$

$y \mapsto \{(pr_1, pk_1), (pr_2, pk_2)\}\}, \emptyset)$

$y = \textbf{Lift}(y, \lambda t.(t, \textbf{switch}(t, z));$

$(\emptyset, \{z \mapsto \{id_1, id_2\},$

$y \mapsto \{(id_1, pr_1, pk_1), (id_2, pr_2, pk_2)\}\}, \emptyset)$

$y = \textbf{MakForwRule}(y);$

$(\emptyset, \{z \mapsto \{id_1, id_2\},$

$y \mapsto \{(id_1, (pk_1, sendout(pr_1)), (id_2, (pk_2, sendout(pr_2)))\}\}, \emptyset)$

$\textbf{AddRules}(y);$

$(\emptyset, \{z \mapsto \{id_1, id_2\},$

$y \mapsto \{(id_1, (pk_1, sendout(pr_1)), (id_2, (pk_2, sendout(pr_2)))\}\},$

$\{(id_1, (pk_1, sendout(pr_1)), (id_2, (pk_2, sendout(pr_2)))\})$

Register;

$(\{(id_1, (pk_1, sendout(pr_1)), (id_2, (pk_2, sendout(pr_2)))\}, \{z \mapsto \{id_1, id_2\},$

$y \mapsto \{(id_1, (pk_1, sendout(pr_1)), (id_2, (pk_2, sendout(pr_2)))\}\}, \emptyset)$

Figure 11: Program 2.

Figure 12: Static Operational Semantics for Two Control programs in ImpNet.

event stored in z. The third statement stores the rule assignment of z in ir as an initial rule assignment. The last statement of the program adds the established rules to the flow tables of switches. Figure 10 shows the static operational semantics of this program using the semantics of the previous section.

The second example constructs forwarding rules based on source IPs of arriving packets and then installs the established rules to flow tables of switch IDs stored in z. This program has the following statements.

$$y = \text{SourceIps};$$
$$y = \text{ApplyLft}(y, \lambda t.(t, \text{port}(t)));$$
$$y = \text{Lift}(y, \lambda t.(t, \text{switch}(t, z));$$
$$y = \text{MakForwRule}(y);$$
$$\text{AddRules}(y);$$
$$Register;$$

The first statement of the program assumes a function *SourceIps* that returns source IPs of arriving packets and stores them in the form of an event in y. The second statement transfers event of y into event of pairs of IPs and port numbers through which packets will be forwarded. The third statement augments values of event in y with switch IDs from the event stored in z. The fourth statement makes a forward rule for each value of the event stored in y. Then the fifth statement stores the rule assignment of y in ir as an initial rule assignment. The last statement of the program adds the established rules to the flow tables of switches. Figure 11 shows the static operational semantics of this program using the semantics of the previous section.

Using a sequence of prohibited IPs (stored in the variable a), the third example constructs firewall rules. Firewall rules are established by first applying *ApplyRit* that applies a map to get the concerned port numbers. Then the map *Lift* is used twice in a row to add switch IDs to items in b and to determine that the action of rules being established is prohibition. Finally the program installs the established rules to flow tables of switch IDs stored in c. This program has the following statements.

$$a = \text{ProhibtedIps};$$
$$b = \text{ApplyRit}(a, \lambda t.(t, \text{port}(t)));$$
$$b = \text{Lift}(b, \lambda t.(t, \text{switch}(t, c));$$
$$b = \text{Lift}(b, \lambda t.(t, \text{prohibt}(t, c));$$
$$b = \text{MakRule}(b);$$
$$\text{AddRules}(b);$$
$$Register;$$

A fourth example, that switches between actions of forwarding and dropping depending on source IPs, can constructed as a combination of the second and third examples above.

The examples shown above demonstrate the importance and robustness of the programming model developed for software-defined networks in this paper. For example the third example above implements a very important concept (Firewalling) of networks in a very compact and accurate way.

6 Future Work

There are many interesting directions for future work. One such direction is to develop methods for static analysis of network programming languages. Obviously associating these analyses with correctness proofs, in the spirit of [18, 19, 20, 21], will have many network applications. Developing denotational semantics [17] for network programs is a very interesting direction for future research. This will increase the trust level of these programs. Another direction for future work is to develop type systems to detect event-errors in the sense of the work in [23]. Implementing the problems of indoor mobile target localization (for wireless sensor networks) [38] and that of CSPs search strategies [45] (for control network programmings) using *ImpNet* are among interesting directions of future work.

7 Conclusion

Software-Defined Network (SDN) is a recent architecture of networks in which a controller device is used to program other network devices (specially switches). This is done via a sequence of installing and uninstalling of rules to memories of these devices.

In this paper, we presented a high-level imperative network programming language, called *ImpNet*, to facilitate the job of controller. *ImpNet* produces efficient, yet simple, and powerful programs. *ImpNet* has the advantages of simplicity, expressivity, propositionally, and more importantly being imperative. The paper also introduced two concrete operational semantics to meanings of *ImpNet* constructs. One of the proposed semantics is static and the other is dynamic. Detailed examples of using *ImpNet* and the operational semantics were also illustrated in this paper.

The proposed language can be used to program many network applications like switch load-balancing. The proposed language can also be realized as a new framework for network-programming that enables applying static and dynamic analysis techniques to network programs.

References:

[1] X. Cao, M. Klusch. Dynamic Semantic Data Replication for K-Random Search in Peer-to-Peer Networks. *NCA*, 2012, p. 20–27.

[2] A. Eftychiou, B. Vrusias, N. Antonopoulos. A dynamically semantic platform for efficient information retrieval in P2P networks. *IJGUC* 3(4), 2012, pp. 271–283.

[3] A. Kamoun, S. Tazi, K.l Drira. FADYRCOS, a semantic interoperability framework for collaborative model-based dynamic reconfiguration of networked services. *Computers in Industry* (CII), 63(8),2012, pp. 756–765.

[4] I. Lerner, S. Bentin, O. Shriki. Spreading Activation in an Attractor Network With Latching Dynamics: Automatic Semantic Priming Revisited. *Cognitive Science* (COGSCI), 36(8), 2012, pp. 1339–1382 .

[5] T. Serbanuta, G. Rosu, and J. Meseguer. A rewriting logic approach to operational semantics. *Inf. Comput.*, 2009, pp.305–340.

[6] B. Loo, J. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative routing: Extensible routing with declarative queries. *SIGCOMM*, 2005, pp.289-300.

[7] Z. Cai, A. Cox, and T. Ng. Maestro. A system for scalable OpenFlow control. *Technical Report TR10-08, Rice University*, Dec 2010.

[8] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: A distributed control platform for large-scale production networks. *OSDI*, Oct 2010.

[9] N. Handigol, S. Seetharaman, M. Flajslik, N. McKeown, and R. Johari. Plug-n-Serve. Load-balancing web traffic using OpenFlow. *Demo at ACM SIGCOMM*, Aug 2009.

[10] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown. ElasticTree: Saving energy in data center networks. *NSDI*, Apr 2010.

[11] R. Wang, D. Butnariu, and J. Rexford. OpenFlow-based server load balancing gone wild. *Hot-ICE*, Mar 2011.

[12] A. Greenberg, G. Hjalmtysson, D. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang. A clean slate 4D approach to network control and management. *SIGCOMM CCR* 35, October 2005, pp.41-54.

[13] M. Casado, M. Freedman, J. Pettit, J. Luo, N. Gude, N. McKeown, and S. Shenker. Rethinking enterprise network control. *Trans. on Networking*. 17(4), Aug 2009.

[14] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: Towards an operating system for networks. *SIGCOMM CCR* 38(3), 2008.

[15] N. Foster, A. Guha, M. Reitblatt, A. Story, M. Freedman, N. Katta, C. Monsanto, J. Reich, J. Rexford, C. Schlesinger, D. Walker, R. Harrison. Languages for software-defined networks. *IEEE Communications Magazine* 51(2), 2013, pp. 128–134.

[16] T. Bain, P. Campbell, J. Karlsson. Modeling growth and dynamics of neural networks via message passing in Erlang: neural models have a natural home in message passing functional programming languages. *Erlang Workshop*, 2011, pp. 94-97.

[17] M. A. El-Zawawy. Semantic spaces in Priestley form. PhD thesis, University of Birmingham, UK, 2007.

[18] M. El-Zawawy. Probabilistic pointer analysis for multithreaded programs. *ScienceAsia* 37(4), 2011, pp. 344-354.

[19] M. El-Zawawy. Detection of Probabilistic Dangling References in Multi-core Programs Using Proof-Supported Tools. *ICCSA* 2013, pp. 516–530.

[20] M. El-Zawawy. Frequent Statement and Dereference Elimination for Distributed Programs. *ICCSA*, 2013, pp. 82–97.

[21] M. El-Zawawy. Flow sensitive-insensitive pointer analysis based memory safety for multithreaded programs. In: Murgante, B., Gervasi, O., Iglesias, A., Taniar, D., Apduhan, B.O. (eds.) *ICCSA*, Part V. LNCS, vol. 6786, Springer, Heidelberg (2011), pp. 355-369.

[22] Mohamed A. El-Zawawy and Adel I. AlSalem. ImNet: An Imperative Network Programming Language. In: Proceedings of The 14th International Conference on Applied Computer Science – Constantin Buzatu (Ed): *ACS* 2014, Modern Computer Applications in Science and Education, pp. 149–156.

[23] M. A. El-Zawawy, N. Daoud. New error-recovery techniques for faulty-calls of functions. *Computer and Information Science*, 5(3),2012.

[24] T. Suzuki, K. Pinte, T. Cutsem, W. De Meuter, A. Yonezawa. Programming language support for routing in pervasive networks. *PerCom Workshops*, 2011, pp. 226–232.

[25] A. Elsts, L. Selavo. A user-centric approach to wireless sensor network programming languages. *SESENA* 2012, pp. 29–30.

[26] C. Monsanto, N. Foster, R. Harrison, D. Walker. A compiler and run-time system for network programming languages. *POPL* 2012, pp. 217–230.

[27] S. Hong, Y. Joung. Meso: an object-oriented programming language for building strongly-typed internet-based network applications. *SAC* 2013, pp.1579–1586.

[28] J. Rexford. Programming languages for programmable networks. *POPL* 2012, pp. 215–216.

[29] H. Arneson, C. Langbort. A linear programming approach to routing control in networks of constrained linear positive systems. *Automatica* 48(5), 2012, pp. 800-807.

[30] B. Loo, T. Condie, J. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. *SIGOPS* 39(5), 2005, pp 75-90.

[31] B. Loo, J. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative routing: Extensible routing with declarative queries. *SIGCOMM*, 2005, pp. 289-300.

[32] N. Foster, R. Harrison, M. Meola, M. Freedman, J. Rexford, and D. Walker. Frenetic: A high-level langauge for OpenFlow networks. *PRESTO*, Nov 2010.

[33] N. Foster,R. Harrison, M. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A Network Programming Language.*the 16th ACM SIGPLAN international conference on Functional programming*,2011 pp. 279–291.

[34] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems* 18(3), Aug 2000, pp 263-297.

[35] S. Egorov and G. Savchuk. SNORTRAN: An Optimizing Compiler for Snort Rules. *Fidelis Security Systems*, 2002.

[36] V. Paxson. Bro: A system for detecting network intruders in realtime. *Computer Networks* 31(2324), Dec 1999, pp. 2435-2463.

[37] M. Dobrescu, N. Egi, K. Argyraki, B. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting parallelism to scale software routers. *SOSP*, Oct 2009.

[38] P. Gao, W. Shi, H. Li, W. Zhou. Indoor Mobile Target Localization Based on Path-planning and Prediction in Wireless Sensor Networks. *WSEAS Transactions on Computers* 12(3), 2013, pp. 116–127.

[39] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM CCR* 38(2), 2008, pp. 69-74.

[40] M. Chen, X. Li, R. Lian, J. Lin, L. Liu, T. Liu, and R. Ju. Shangri-la: Achieving high performance from compiled network applications while enabling ease of programming. *PLDI*, Jun 2005, pp 224-236.

[41] V. Bhanumathi, R. Dhanasekaran. Energy Efficient Routing with Transmission Power Control based Biobjective Path Selection Model for Mobile Ad-hoc Network. *WSEAS Transactions on Computers* 11(11), 2012, pp. 407–417.

[42] M. Cristea, C. Zissulescu, E. Deprettere, and H. Bos. FPL-3E: Towards language support for reconfigurable packet processing. *SAMOS*, Jul 2005, pp 201-212.

[43] The Open Networking Foundation, Mar 2011. See http:// www.opennetworkingfoundation.org/

[44] J. Harding. Decidability of the Equational Theory of the Continuous Geometry CG(F). *J. Philosophical Logic* 42(3), 2013, pp. 461–465.

[45] E. Golemanova. Declarative Implementations of Search Strategies for Solving CSPs in Control Network Programmings. *WSEAS Transactions on Computers* 12(4), 2013, pp. 174–183.