Mapping Applications Intents to Programmable NDN Data-Planes via Event-B Machines

OUASSIM KARRAKCHOU, (Graduate Student Member, IEEE), NANCY SAMAAN, (Member, IEEE), AND AHMED KARMOUCH, (Member, IEEE)
School of Electrical and Computer Engineering, University of Ottawa, Ottawa, ON K1N 6N5, Canada
Corresponding author: Ouassim Karrakchou (okarr102@uottawa.ca)

This work was supported by a Discovery Grant from the Natural Sciences and Engineering Research Council of Canada.

ABSTRACT Location-agnostic content delivery, in-network caching, and native support for multicast, mobility, and security are key features of the novel named data networks (NDN) paradigm. NDNs are ideal for hosting content-centric next-generation applications such as Internet of things (IoT) and virtual reality. Intent-driven management is poised to enhance the performance of the offered NDN services to these applications while reducing its management complexity. This article proposes I2DN, intent-driven NDN, a novel architecture that aims at realizing the first step towards intent modeling and mapping to data-plane configurations for NDNs. In I2DN, network operators and application developers express their abstract and declarative content delivery and network service goals and constraints using uttered or written intents. The intents are classified using built-in intent templates, and a slot filling procedure identifies the semantics of the intent. We then employ Event-B machine (EBM) language modeling to represent these intents and their semantics. The resulting EBMs are then gradually refined to represent configurations at the NDN programmable data-plane. The advantages of the proposed adoption of EBM modeling are twofold. First, EBMs accurately capture the desired behavior of the network in response to the specified intents and automatically refine it into concrete configurations. Second, EBM’s formal verification property, referred to as its proof obligation, ensures that the desired properties of the network or its services, as defined by the intent, remain satisfied by the refined EBM representing the final data-plane configurations. Experimental evaluation results demonstrate the feasibility and efficiency of our proposed work.

INDEX TERMS Event-B machines, intent-driven networking, named data networks, programmable data-planes.

I. INTRODUCTION
Named data networks (NDNs) [1], [2] and intent-driven networking (IDN) [3], [4] are two orthogonal research paradigms that aim at revolutionizing the current use of networks from conventional communication services into integral components of next-generation applications. Examples of these applications include time-sensitive, content-centric, and dispersed applications that allow humans to interact seamlessly with virtual objects within the context of virtual and augmented reality. Industrial automation functionalities built on top of sensors and connected machines represent another example of these applications. On the one hand, NDNs facilitate building advanced applications by shifting the application developers’ focus away from address- and location-centric communication and towards a simplified content-centric one. On the other hand, IDN allows network operators and hosted application developers to describe what is required from the network at a high level of abstraction without being concerned about how these requirements should be implemented at the network data-plane [4].

NDNs are designed to deliver contents that are uniquely identified using hierarchical naming structures such as the Uniform Resource Identifiers (URIs) [2]. Contents can be conventional data components such as files, video clip chunks, or books but can also represent sensor readings or exchanged commands between machines. NDNs operate using two packet types: interest packets (Ipkts) and data packets (Dpkts). A content consumer (e.g., a user device) sends an Ipkt containing the name of the required content in the network. Each switch then serves the Ipkt by either forwarding it along a path to the content producer or to a nearby router that is already storing the requested contents in its cache. A content producer or a router storing the content
then replies with a Dpkt containing the requested content. The Dpkt follows the reverse path of the Ipkt until it reaches the Ipkt sender.

NDN’s simplified mechanism natively supports multicast services while eliminating the well-known IP addressing problems, such as address scalability and user mobility. Its location-agnostic communication also facilitates hosting distributed applications executing on virtual resources [5]. These advanced functionalities are attributed to a more complex NDN data-plane. NDN switches must manage forwarding information bases (FIBs) storing content name prefixes, special pending interests tables (PITs) logging unsatisfied requests, as well as content stores (CSs) that can cache received contents. These tables transform NDN switches into state-aware devices that can make adaptive packet processing, forwarding, and content caching decisions in addition to traditional traffic engineering and network services such as traffic shaping and monitoring.

Network operators are envisioned to take advantage of NDN switch functionalities to offer novel per-application, per-content, or per-consumer highly customized network services such as time-sensitive delivery using prefetching and caching, semantics-based forwarding, and content encryption and decryption [6]. This vision is motivated by emerging technologies, such as software-defined networks [7], that succeeded in separating the network control functionality from that of the data-plane packet forwarding process. More recently, the emergence of switch programming languages, such as P4 [8], has enabled the notion of programmable data-planes (PDPs). Using these languages, the control-plane can continuously configure and fine-tune the switch behavior with respect to packet parsing and processing [9].

Despite these advances, operators are still limited by the current network management tools to direct the installation of per-flow or per-path switch configurations. These tools may require error-prone manual configuration and policy validation [4]. In addition, control-plane functionalities (e.g., routing, traffic engineering, and congestion control) still require manual parameter setting and have a network-wide service focus that lacks the needed per-application customization. Finally, these tools provide no direct means for application developers or users to directly define their network service requirements in a declarative manner.

The emerging concept of IDN attempts to bridge the gap between network management complexity and the emerging network service demands on one side and advances in data-plane programmability on the other [3]. The main premise of IDN is to allow operators and application developers to describe what is expected from the network serving the applications but not how that behavior is implemented using intents [10]. IDN tools can then automatically “convert, verify, deploy, configure and optimize” [4] the network to satisfy these intents. The realization of IDN necessitates addressing three main challenges: first, the development of expressive intent and network state models. Second, the realization of new mechanisms to automate intent validation and mapping to data-plane configurations. Third, novel intelligent machine learning-based techniques must be developed to allow the network to continuously self-adapt and self-heal to maintain the satisfaction of these intents [11].

This article addresses the first two challenges described above. We consider a single domain NDN with programmable switches that each can process packets using a chain of stateful match-action tables (MATs) (e.g., switches based on P4 [9] or those supporting program-based forwarding strategies [12]). We propose a novel intent-driven NDN (I2DN) architecture that models and captures high-level intents and transforms them into configurations for the programmable NDN data-plane. In I2DN, intents are first captured as uttered or written sentences. These are tokenized and classified using preexisting intent templates. A slot filling procedure is then employed to extract a set of intent parameters from the uttered words. The output from this phase is then translated using Event-B modeling into abstract Event-B machines (EBM) [13] which provide abstract descriptions of the desired network behavior to satisfy the given intents. Each EBM describes a desired behavior as a set of events acting on an abstract state representing the network. Abstract EBMs are then refined using existing tools, such as Rodin [14], to gradually introduce network-specific configurations implementing the desired behavior until a concrete EBM is developed. The concrete EBMs closely resemble the structure of the programmable MATs in the data-plane. Hence, they are transformed or compiled into an equivalent data-plane behavior satisfying the intent.

The adoption of EBM modeling serves two main purposes. First, the highly abstract model of the EBMs describing the intents represents ideal means to capture the intent goals. Meanwhile, refinement, a key feature of EBM modeling, allows for the gradual mapping of these hardware and software-independent abstract EBMs towards the concrete EBMs representing the corresponding data-plane configurations. Second, Event-B is also a formal method to design EBMs that are correct by construction. I2DN benefits from this feature by formally representing an intent requirements and constraints on the network states by defining strict rules referred to in the EBM as invariants. For a machine to be correct, i.e., performing as intended, these invariants must always be preserved after every event and refinement operation. These verification steps are referred to as proof obligations and are carried-out using automated tools such as Rodin [14]. To this end, the main contributions of this article can be summarized as follows:

1) We develop a general framework for the lifecycle management of intents within the context of NDNs and analyze the main challenges for its realization. We then propose I2DN, a novel architecture that focuses on modeling and mapping NDN intents into data-plane configurations.

2) Within I2DN, we define a novel networking intent model that is inspired by existing virtual assistants.
3) We propose a novel intent-to-data-plane configuration mapping process using Event-B modeling. The proposed work demonstrates how EBM modeling language and refinement tools can be used efficiently to automate the steps of intent processing, validation, and translation to correct network and domain-dependent configurations.

The remainder of this article is organized as follows; Section II presents the main concepts of NDNs and discusses how programmable data-planes are realized in the context of NDN. Section III introduces IDNs, explains their relevance, and surveys the related Literature. In Section IV, we provide an overview of our proposed mapping architecture. Section V is then dedicated to describing the adopted models and their mapping steps. Simulation results are presented in Section VI. Section VII discusses some open research issues for I2DN. Finally, Section VIII concludes the article and presents planned future work.

II. NDN BACKGROUND AND RELATED WORK

In this section, we first provide a brief review of NDNs’ data-plane functionalities and discuss current progress with respect to achieving programmability at that plane.

A. NDNs AND SWITCH CONFIGURATIONS

NDNs are centered around the delivery of contents that are uniquely identified using a hierarchical content naming format (e.g., /com/youtube [2]). Rather than IP addresses, NDN components, including network switches, servers, connected sensors, machines, and user devices, are identified by semantically meaningful names. In turn, any device in an NDN network can act as a content producer, a consumer, or a packet forwarder simultaneously [15].

As shown in Fig. 1, to request contents, a consumer generates an Ipkt that is sent to an NDN switch. An Ipkt contains the requested content name as well as optional metadata to specify any additional constraints on the delivered content, such as its version, freshness, or publisher. Each Ipkt is also uniquely identified using a randomly generated nonce value that must be added before the Ipkt is dispatched to the network. Additionally, an Ipkt can include forwarding hints instructions specifying a particular routing path as well as any arbitrary metadata that can be used to parameterize the delivered contents. To forward an Ipkt, each device, including the user’s device and the switches, looks up its forwarding information base (FIB) to find the longest prefix match to the Ipkt content name and a corresponding list of candidate ports for Ipkt forwarding. Finally, configured forwarding strategies define additional rules (e.g., all ports, least occupied port, or first port in the list) controlling the final forwarding action for the Ipkt.

In contrast to IP-based forwarding, the forwarding of Ipkts is stateful: when a switch forwards an Ipkt, it is stored in a pending interest table (PIT) along with its source port until the interest expires or is satisfied. If another Ipkt with the same content name is received, the switch adds the source port of the new Ipkt to the matched entry in the PIT. This allows the switch to store the states of all currently served interest requests while avoiding overloading the network with redundant requests of the same content. In addition, PITs facilitate multicast services and loop-free multipaths. Loss recovery with minimal latency is also easily achieved by controlling timeouts for the PIT entries.

As shown in Fig. 1, when a content producer receives an Ipkt, it replies with a Dpkt sent on the Ipkt source port. A Dpkt contains the requested content and its name. The publisher ensures authentication of the data by adding a signature field along with any additional tags (e.g., published information, content version, and creation time) that can be stored in the metadata. When a switch receives a Dpkt, it looks up its PIT to forward the Dpkt along the reverse paths of the corresponding Ipkt and then erases that entry from the PIT. When a host receives a Dpkt, it uses its PIT to forward it to the correct application interface. Mobility of consumers or producers is inherently treated in NDNs since location-dependent IP addresses do not identify packets. For instance, a moving consumer can resubmit a request to desired contents for an expired Ipkt. NDN switches can also add forwarding hints to Ipkts to guide them to a new producer location.

An NDN switch also contains a content store (CS) to cache forwarded Dpkts according to a specific caching strategy. For example, the switch adjacent to the first consumer in Fig. 1 caches the received Dpkt and sends it to the second consumer in response to a new request. Thus, cached Dpkts used to reply to multiple Ipkts can significantly reduce content delivery latency in NDNs.

B. DATA-PLANE PROGRAMMABILITY

The data-plane layer of NDN is stateful by design since records of all pending Ipkts are stored in the PITs. Furthermore, with CSs, switches can employ different caching strategies to reduce content delivery latency. In addition, switches parse content names and employ them for routing. Finally, the original NDN design [1] envisioned fully programmable and adaptive forwarding strategies that can be implemented using programming algorithms. These design features allow NDNs...
to support the development of new network services (e.g., content ordering, freshness guarantees, semantics-based forwarding, authentication, and/or publish/subscribe related services). However, most of these features remain conceptual at the design level, with little progress towards their realization on current switches. Meanwhile, existing literature has focused on the design issues of various NDN functionalities mainly such as routing [16], forwarding [2], [17], and caching [18], [19].

Recently, several research efforts have focused on the adoption of the novel paradigm of software-defined networking (SDN) for the efficient management of NDNs [7], [20], [21]. In SDN, network control logic and algorithms are executed at the control-plane, which then communicates a set of forwarding rules directly to the switch data-plane using protocols such as OpenFlow. However, existing solutions have mostly focused on implementing specific services such as routing [22], traffic management [23] and adaptive caching [24]. In these approaches, the controller achieves limited reconfigurability of the NDN switch data-plane using OpenFlow [25].

In previous work [9], the authors developed a novel NDN programmable data-plane (PDP) architecture that takes advantage of P4, a switch behavior programming language [8]. The proposed work allows a controller at the control-plane to define, install and update, at run-time, customized P4 programs to realize a suite of network services. In the proposed work, each programmed switch first parses Ipkt and Dpkt headers and collects additional metadata such as its source port or the size of its ingress queue. The switch then processes the packet according to a sequence of Match-Action Tables (MATs) that are programmed according to the controller’s specified P4 programs. Programmed instructions may send the packet on specific ports looked up using its FIB or PIT, as well as drop, recirculate, or clone the packet. The switch may also collect and store different statistics about the packet. The packet may also be sent to the CS to be stored if it is a Dpkt or replied to if it is an Ipkt. The proposed work then demonstrated how these functionalities are used to offer different services in the data-plane. Examples of these services include traditional ones such as admission control, load balancing, security using firewalls, and differentiated content-delivery services. Other examples include novel services such as geographical gating and caching.

In this article, we take advantage of the developed PDP architecture and focus on the problem of translating high-level intents to these customized PDP configurations.

### III. INTENT-DRIVEN NETWORKING

The main premise of IDN is to have networks that are easier and simpler to manage and customize to individual applications and/or industries [11]. IDN allows operators to describe, at a high level of abstraction, the desired business goals as well as how customized network services should behave to serve different applications. IDN can also be employed by application developers to interact directly with the hosted network to specify their required service customization. This section describes the main intent lifecycle management functionalities and discusses the main contributions towards their realization based on the model defined by the IRTF Network Management Research Group of the IETF [11].

#### A. INTENT LIFECYCLE FUNCTIONALITIES

Within the context of IDN, an intent describes a goal, a constraint, or a desired outcome to be met by the network [26]. The authors in [26] define three main intent types: (i) customer- or application-service intents that describe the desired service quality for a given customer or application (e.g., customers should receive application A videos with high quality and a staleness not exceeding one minute); (ii) network-service intents describe services that are offered by the network (e.g., content delivery services should have a maximum latency of 30 ms); (iii) strategy intents describe a desired goal from the perspective of the overall network operation (e.g., reduce overall energy consumption or maintain bandwidth utilization levels and cache occupancy below a given threshold). Intents can also be classified according to their lifecycle as either persistent (e.g., all users of a given application receive the highest video quality) or transient (e.g., remove all cached contents of a given application from the network).

Fig. 2 depicts the main processing functionalities during the lifecycle of an intent. The figure builds on the IETF standard model [11] and includes two main phases: pre-production and production. During the first phase, the network operator defines the set of intents that the users can employ. Then, depending on the level of automation in the IDN [27], the operator may optionally associate with each intent an intent handler to define the abstract actions that are taken by the network to fulfill the given intent. These handlers can range in complexity from predefined rules to self-reasoning agents that learn and refine the intent handling using feedback from the network. These handlers/rules will aid the intent translation process during the production phase.

The first functionality in the production phase involves ingesting the intents from the users. These users can be network administrators, application developers, or end-users. This step takes place using different text- or voice-based interfaces to type or utter the intents, respectively. Advances in speech recognition and natural language processing allow for the realization of this step [28]. Moreover, the authors in [11] envision this process to eventually include an open dialog between the user and the IDN system in order to aid the user to articulate and clarify the intent gradually.

Once ingested, the intent lifecycle management involves the realization of functionalities that belong to one of two categories, namely, intent fulfillment and assurance [11]. Functions in the first category ensure the realization of the required network configurations to satisfy the intent. Meanwhile, assurance functionalities validate the intents, identify any potential conflicts with already existing ones, and ensure...
that the corresponding switch configurations realize the goals of the intents and do not drift away from these goals over time.

The first step in intent fulfillment involves identifying the ingested intent. In this step, the intent is rendered in a format that the IDN system can process. This step includes identifying the type of the intent, its application scope, its goals, and/or desired outcomes. It also parses the intent to identify any semantics that the user has provided within the ingested intent (e.g., a specific content, time, or service name). The outcome of the identification process is fed to the translation module which maps the intent into actions, management operations or services, as well as network configurations. Any predefined intent rules or handlers that were defined in the pre-production phase can be used as aids to this step. The final stage in the fulfillment of an intent is to translate that intermediate representation into device-specific configurations. The orchestration of the configurations of different devices in the network to respond to different intents also represents an important component of this final stage.

Intent assurance functionalities ensure that the applied network configurations comply with the user intents. These functionalities include intents conflict detection and resolution as well as the assurance that the implemented configurations satisfy the intents. The first step of the intent conflict detection process takes place before the network configurations are deployed. Then during network operation, the traffic is monitored and analyzed to ensure the intents goals are satisfied.

IDN systems are anticipated to be augmented with machine learning (ML) capabilities [4] that can enhance the performance of various IDN functionalities using learnt experience. For example, as will be shown in our proposed work, intent identification tools may employ ML algorithms to enhance the process of understanding the user input. Similarly, a ML-based translation module can refine its mapping decisions based on the network feedback concerning previous configurations. Finally, ML can be used to monitor and analyze the network feedback and take appropriate actions to correct the data-plane configuration when the network performance shifts away from the intent goals [29].

Using the above framework, we can identify three main areas of research. First, the development of formal models for representing intents and intent handlers is a key step towards the automation of IDN systems. Second, the development of efficient mechanisms for intent translation into network configurations as well as intent conflict detection and configuration validation before deployment is another challenge. Finally, the last challenge concerns the addition of the necessary intelligence for each IDN system functionalities to ensure its full automation. In our proposed architecture, we focus on the first two of these challenges. Hence, in the following section, we review the Literature with respect to intent modeling, translation, and validation.

B. RELATED APPROACHES FOR INTENT MODELING AND TRANSLATION

Existing network data models such as the management information bases (MIBs) and YANG (yet another next-generation) were developed specifically for low-level device configuration [30]. They are accompanied by a suite of client-server protocols such as the simple network communication protocol (SNMP) and the network configuration protocol (NETCONF) to interact with and configure devices. While they provide a good abstraction for device configurations, they are not suitable for representing the high-level abstraction of network intents.

While recent research efforts have proposed several novel intent models, they have been mostly focused on defining intents that directly capture desired network or service configurations rather than abstract or declarative user or operator goals. For example, one of the earliest approaches for intent modeling is the model built within the SDN-based Open Network Operating System (ONOS) [31]. The model defines a set of predefined connection-oriented intents (e.g., topology, end-points connection, or service chain intents) and then provides a one-to-one mapping of these intents to network policies. Similarly, the IETF NEMO project and its extension defined in [32] focus on intents relating to network operations, such as selecting or changing a routing path. Other approaches utilize intent models built as extensions of the Topology and Orchestration Specifications for Cloud Applications (TOSCA) model [33]. However, they are also limited to direct mapping of network-oriented low-level intents into policies. Chopin [34] is another framework for specifying intents for cloud resource usage between endpoints. It uses a fixed intent template that defines the desired traffic source and destination as well as the required resources between these endpoints. The authors in [35] develop a novel intent definition language for applications hosted on IP networks. In their model, intents must clearly identify the two communicating endpoints and the desired data-plane service (e.g., drop heavy hitters), which is then configured statically in the data-plane. In a similar manner, an intent model was developed in [36] to describe flow-rule intents.
TABLE 1. Summary of the analysis of intent-based solutions in the Literature.

| Intent-Based Solution (authors) | Intent Model/Scope | Applicability Domain |
|---------------------------------|-------------------|----------------------|
| Han et al. [31]                 | Virtual network model with middleboxes | Define virtual networks |
| Tsuchaki et al. [32]            | Event-Condition-Action based on network resources (e.g., switch bandwidth) | Reactive network path change based on bandwidth usage |
| Khan et al. [33]                | 5G service model and Quality of Service (QoS) requirements | Automate 5G network configuration and slicing |
| Heorhadi et al. [34]           | Network resources cost and constraints between endpoints | Network resource management and optimization |
| Riffati et al. [35]            | P4 services between endpoints for a specific traffic flow | Create a combined P4 code containing multiple services |
| Singh et al. [36]              | QoS requirements | Data dissemination in vehicular edge networks |
| Alamei et al. [10]             | Service requirements | Cloud CDN |
| Proposed work                  | Name-based service requirements and constraints represented using EBM s | NDNs |

In summary, existing network intent models are limited to describing communication-oriented requirements rather than aiming at capturing the operator or application goals from the underlying network. The majority of these existing models assume that the served applications have a detailed knowledge of the network topology and the exact configurations of the resource demands for their traffic flows. In other words, they identify network configurations using low-level vocabulary (e.g., allocated bandwidth between two endpoints). A detailed comparison of existing IDN models and their limitations is presented in [10].

In contrast to the aforementioned models, highly expressive and well-developed intent models were developed for software applications such as those used by personal assistants [37], [38]. Moreover, intents capture and interpretation using these models have been addressed extensively in the field of natural language processing [39].

The majority of existing solutions in the Literature for intent to network configuration focus on the direct mapping of intents into policies [11], [32]. However, one of the main limitations of this approach is that the rigid modeling of policies as events-condition-actions fails to capture intent goals except in the context of predefined services such as network slicing [40]. A different approach is used in [34] where intents are translated directly into optimization problems for resource assignment and allocation. Overall, the Literature is limited to approaches that map intents to policies or limited direct network configurations.

Table 1 presents a summary of the intent models and domains of applicability of the major existing solutions in the Literature. Most of these solutions are domain-specific, and, hence, provide an intent model that captures requirements specific to a certain use case. Additionally, these solutions all apply only to a topology-centric IP-based network. To the best of the authors’ knowledge, the proposed work is the first attempt to build an intent model and an intent-to-data-plane mapping mechanism with a particular focus on NDNs. As NDN names are generic and can identify both contents and network resources, an NDN-based intent model offers a higher-level of abstraction compared to IP-based models. Thus, application developers can define high-level custom network services applied to their contents and flows without any prior knowledge of the underlying network topology or endpoints.

IV. PROPOSED I2DN ARCHITECTURE

A. I2DN NETWORK MODEL

As shown in Fig. 3, the goal of I2DN is to receive intents from network operators or application developers and then translate them into a programmable NDN data-plane configuration. The target network contains a single domain managed by a single controller. We further require that the switches in the NDN data-plane implement stateful programmable Match-Action Tables (MATs) that can process packets according to custom rules. These MATs can be semantically represented as a set of rules of the form if (conditions) then actions, where the conditions and actions apply to packet fields (e.g., content name), switch metadata (e.g., queue length or output port),
or custom saved states. Furthermore, we assume that access to the CS is controlled by the MATs as shown in Fig. 3.

The Literature contains several data-plane architectures that meet these requirements and can thus be used with I2DN. We can cite P4 switches [8], OpenState [41], or our proposed ENDN architecture [9]. Traditional NDN switches can also be used if they allow the creation of new custom stateful forwarding strategies. The stateful programmable data-plane allows highly dynamic per-packet forwarding decisions to be executed directly at the data-plane with little involvement from the controller. As a result, communication between switches and the controller for data-plane configuration is carried out only when a new intent is requested: every intent is translated to stateful MAT entries in the data-plane.

### B. OVERVIEW OF I2DN

Fig. 4 provides a schematic description of the main components of our proposed I2DN architecture. As per the model described in Sec. III, the processes of I2DN operate in two phases: production and pre-production. The production phase corresponds to the intent to data-plane configuration mapping process that is executed each time a new intent is uttered. On the other hand, the pre-production phase consists of defining all the different mapping rules used during the production phase. For instance, the different types of intents that the users can request is defined by the operator in a library of intents templates during the pre-production phase. These intent templates are related to a service or a network strategy. Examples of strategy intents are: to forward a given list of contents to certain subscribers, to cache contents belonging to a particular namespace for a specific duration or to distribute requests equally among several producers. Examples of service intents are: to maintain average utilization of a server to a certain level or to create three classes of services for contents. Intents also have parameters called slots (e.g., a content namespace or a traffic threshold).

The production phase consists of an intent processing workflow containing three main steps: identification, translation, and configuration. These steps are closely related to the stages of a generic IDN intent lifecycle, as shown in Fig. 2. The validation process is done in parallel with the translation and configuration steps using the proof engine of the Event-B formal method.

During the identification step, intents are captured using a chat interface [42] or with the help of a smart assistant similar to Amazon’s Alexa [37]. The intent detection and slot filling [43] operations are then performed. In this step, an intent is identified by contrasting it against the built-in intents from the intent library, and a list of label-value pairs representing intent slot parameters is generated (e.g., time intervals, content names, or producers IDs).

Once the slot labels and values are obtained, they are fed into the first module of intent translation: the abstract Event-B machine (EBM) generation. Every intent template is associated with an abstract EBM during the pre-production phase. This EBM contains an abstract implementation of the desired network behavior to fulfill the intent. Event-B [13] is a formal method that allows developers to model a discrete system control problem using a set of state variables in an EBM. Constraints, called invariants, are then added to the possible values of the state variables to represent the expected system behavior when the problem is solved (e.g., a counter can never reach a certain threshold). Events acting on the state variables are then created and proven to be compliant with the constraints, thus resulting in an event-based algorithm that solves the control problem. Event-B can thus create programs that are proven to be correct by construction using its proof engine [14]. Our architecture uses Event-B to model the programmable network behavior in response to each desired intent. EBM events follow the if (condition) then action semantic. This representation facilitates the refinement of the abstract machines into corresponding Match-Action Table (MAT) rules in the data-plane. In this case, EBM state variables correspond to packet headers, traffic statistics, switch values (e.g., queue size), packet metadata (e.g., packet source and destination), and in-network custom saved states (e.g., the last measured RTT), and thus correspond to the different inputs and outputs of the network.

In the abstract EBM, intent slot values are mapped to EBM parameters, and the semantics of the intent result in several invariants that ensure that the EBM implements the required intent behavior. Once the abstract EBM is instantiated with the slot values, it is refined using several refinement patterns [44] defined in the pre-production phase until a final EBM, called concrete EBM, is reached. EBM refinement is an essential part of the Event-B method: it gradually adds more details to the EBM while ensuring the invariants are always met until the problem is completely solved. The main goal of the refinement step is to transition between two different EBM representations. The abstract EBM representation is high-level and allows the intent requirements to
be defined conveniently using abstract variables. On the other hand, the concrete EBM representation is switch-dependent and thus close to the data-plane MAT structures. As a result, the refinement patterns map abstract EBM variables and events into concrete EBM constructs to adapt to the network capabilities. For instance, a load balancer intent can balance the load between two producers using a specific load distribution algorithm (e.g., round-robin, congestion-aware, or based on the source region of the packets). The abstract EBM would then contain the generic load balancing algorithm and an abstract variable specifying the load distribution algorithm to use. On the other hand, the concrete EBM would contain the full implementation of the load balancer with the load distribution algorithm in the case of a P4 network, or an action to forward the packets to a load balancer middlebox implementing the required load distribution algorithm in a more traditional network. The proof engine is executed during every refinement to ensure that the refined EBMs do not violate the invariants of the abstract EBM. Hence, the concrete EBM is proved to be compliant with the intent requirements set at the abstract EBM level.

Once the concrete EBM corresponding to the intent has been generated, it is processed by the EBM analyzer module. The main goal of this module is to translate the concrete EBM into programmable MAT entries. However, as multiple intents can be configured in the network, we first need to check that these intents do not result in conflicting data-plane configurations. Therefore, the EBM analyzer first performs consistency checks among multiple intents. More precisely, through the composition of different EBMs representing different intents [45], we can ensure that the invariants of an EBM are not violated by the processing done in another EBM. Hence, we can verify that a new intent does not conflict with existing ones. Once the concrete EBM passes the consistency checks, it is translated into a stateful MAT program represented in a model that is compatible with the underlying network, such as a custom forwarding strategy [12] or a P4 program [9], [46]. Finally, it is worth noting that some EBM variables are mapped into the execution of generic control-plane functionalities (e.g., a routing scheme to find the shortest path, or an optimal network function placement algorithm).

The following sections provide a brief description of our proposed models and intent lifecycle functionalities.

V. PROPOSED INTENT LIFECYCLE
In this section, we describe in detail the different steps of the intent lifecycle of our I2DN architecture. Table 2 contains a summary of the different mathematical variables used in this section.

### A. INTENT CREATION AND IDENTIFICATION
In our model, at an abstract level, an NDN can be regarded as a blackbox that provides end-points (e.g., users, devices, and applications) with customizable contents. Customization includes various delivery patterns (request/receive, publish/subscribe, notifications, etc.), content processing services (e.g., encryption, filtering, and synchronization of multiple streams) as well quality guarantees (e.g., reliability, delivery speed, and latency). Furthermore, it provides additional delivery services (e.g., access control, caching, request filtering, load balancing, geo-gating, and delivery quality assurance). The network blackbox also provides monitoring (e.g., reporting the number of requests from a certain user) and event-reporting (e.g., reporting an alarm when the number of content requests in a geographical area exceeds a given threshold) services. From the perspective of the network operator, the network blackbox is composed of a number of abstract services (e.g., content request/response handlers, content filtering, firewalls, and access control) that act on resources (e.g., consumers and producers lists, content namespaces, abstract communication channels to consumers, producers, and contents or caches) that must be configured in order to satisfy the requirements of the offered services.

These requirements are defined as intents that are instances of intent templates. The latter are created by the network operator and are stored in an intents library during the pre-production phase. They are defined using semantic frames [39], [47]. Each frame, or intent template, contains a unique intent name $n$ and a set of entities, referred to as slots that are placeholders for the values of attributes needed to describe the intent. The intent template also provides a set of different example utterances that the intent owner can use. These samples can be communicated to the application developer as hints.

Formally, an intent template is identified by its name $n$ and defines different sequences $s_1, s_2, \cdots$ of slot labels from a set $\mathcal{L}$ such that $s_i = (l_{i1}, l_{i2}, \cdots)$. Each slot label $l \in \mathcal{L}$ describes an object that the users may mention in the intent. Fig. 5 depicts three examples of different intent templates. The first is an intent to describe a load balancing mechanism that an application developer can request. The template indicates the set of slot labels with their types that can be used in that intent (e.g., $cn$, $cl$, and $pl$). The possible sequences of slots are defined by the uttered samples. For example, in the first uttered sample: `distribute the received requests`
for cn using mechanism between p1 and p2”, indicates that the expected slot labels are $s = \{cn, mechanism, p1, p2\}$. The second intent template in the figure describes an intent to cache contents in the network when they satisfy certain properties (e.g., cache contents generated by producer $p1$ in the last hour). Finally, the third template describes an intent requesting to block or report heavy hitters (i.e., consumers who send many Ipkts to a given type of content) in a certain region.

At production time, users utter an intent to describe the desired outcome guided by the samples of uttered intents. The identification module tokenizes the intent into words $w = (w_1, w_2, \cdots)$ that are processed in two steps: intent classification, i.e., mapping the uttered words to the correct intent $n$, and a second phase of slot filling that identifies a corresponding sequence $s_i = (l_1, l_2, \cdots)$ and a corresponding subset of the tokenized words stored in the vector $v_i = (w_1, w_2, \cdots)$ storing the corresponding values of the slots. For example, using the first intent template in Fig. 5, when the user utters “Producer1 and Producer2 should serve Video between 3:00pm to 5:00pm”, the identification module’s output is the intent template name LoadBalance-Action, the slot labels sequence $s = \{p1, p2, cn, t\}$, and the slot values $v = \{\”Producer1\”, \”Producer2\”, \”Video\”, \”3:00pm to 5:00pm\}\}$. It is worth noting that slot values correspond to abstract values that can later be mapped to concrete network-specific values. For instance, the Video slot value corresponds to a content name prefix identifying all the video contents of a specific application in the previous example.

We adopt open-source machine learning-based tools, such as DeepPavlov [48] in this phase. Models of the intents are first defined and stored as JSON objects data sets. The tool is then trained using a graphical user interface until it can correctly identify the intents. When a new intent template is added, the system is retrained to recognize the intent. The outcome of the intent identification phase is a selected intent and a list of slot labels and their values that are then passed to the intent translation phase.

B. EBM TEMPLATES AND INTENT TRANSLATION

We will first describe the abstract EBM templates that the operator creates for each intent and slot sequence. As shown in Fig. 6, we implement an intent behavior in Event-B using two components: a context $C$ and an abstract machine $M$.

The context $C$ defines the relatively static state of the network and is shared by all the machines. On the other hand, every machine implements the behavior of a specific intent. As shown in Fig. 4, the network context is created during the pre-production time but can be updated during production. In Event-B, the context is used to define new data types that are associated with the variables representing the state of EBMs [13]. In our architecture, we thus use the context to represent the types of different resources and objects that are available or can be manipulated in the network. Examples are producers, consumer regions, content namespaces, or scheduling algorithms. Fig. 7 shows the Event-B code of a network context which contains three sections: Sets, Constants, and Axioms. Hence, the context can be modelized by the set of sets $C(S, C, A)$. Here, $S$ lists all the types (i.e., the categories of objects or resources that comprise or interact with the network). The constants set $C$ stores possible elements of the sets in $S$ (e.g., the possible content producers). Here constants can also refer to names of algorithms or control-plane mechanisms that can be resolved during refinement. For example, the $LoadBalanceAlgorithms$ set stores the constants $RoundRobin$ and $WRR$ that correspond to different scheduling algorithms for a load balancer.

```
FIGURE 5. Examples of built-in intent templates.
```

```
FIGURE 6. Examples of abstract EBMs.
```

```
FIGURE 7. An Event-B context.
```

```
Finally, the axioms set \( A \) is used mainly to link constants to their set (e.g., \( axm1 \) in Fig. 7). But it is worth noting that axioms can also be used to specify properties of sets and constants (e.g., every content namespace must have at least one producer).

A machine template \( \mathcal{M} \) contains the implementation of an intent behavior. Table 3 summarizes how intents are mapped to EBMs. At the level of the intent, the network is seen as a blackbox whose expected outcomes are specified. However, in the EBM, we go inside this blackbox and model how the network processes packets to satisfy the intent. In NDN, the network processes two types of packets: Ipkts and Dpkts. Hence, our EBMs specify the stateful treatment of Ipkts and Dpkts inside the network. More precisely, an EBM models an NDN network and its possible packet processing actions using a set of variables \( V \). The variables have a type that can either be a native type (e.g., boolean or integer) or one of the new types defined in the context \( C \).

The EBM variables can be classified into four categories: packet variables, flow variables, abstract variables, and slot parameters. Packet variables correspond to any data specific to a single packet. Hence, they are used to represent header fields (e.g., content name), individual packet forwarding actions (e.g., drop or forward to a specific destination), or metadata (e.g., queue priority, received timestamp, or source region). Packet variables are thus reinitialized each time the network receives a new packet. On the other hand, flow variables represent stateful information that is kept in the network. Examples are data managed by stateful algorithms (e.g., number of packets sent to a specific destination) or contents cached in the network. Abstract variables are only allowed at the level of abstract EBMs and correspond to parts of the packet processing treatment that have not yet been specified in detail. For instance, an abstract EBM may have an abstract variable representing the result of a congestion detection mechanism without detailing how this mechanism works. This abstract EBM would then specify how to process packets in case of congestion based on the value of this abstract variable. The refinement process eliminates the abstract variables by replacing them with the corresponding algorithms. The operator has the complete freedom to decide on the abstraction level that is represented by these abstract variables. A higher level of abstraction will provide more flexibility to adapt to different network domains and capabilities at the expense of refinement steps. Finally, slot parameters are used to make an EBM generic by allowing its behavior to be parametrized.

Packet processing actions are represented in EBMs by a set of events \( E \) that act on the variables \( V \). The events have an \( \text{if} \) (condition) \( \text{then} \) action semantic, where both the condition and actions are relative to the variables \( V \). Hence, an event \( e \in E \) can be formally modeled as a conditional statement: \( e := \text{if} (G_e(V)) \text{ then } A_e(V) \). The event guard \( G_e \) contains a list of logical conditions on the values of the EBM variables \( V \) that can trigger the event. On the other hand, the event action \( A_e(V) \) specifies how variables are modified when \( e \) is executed. Hence, each event that is triggered brings the network from one state to another state. The possible states of the machine are restricted by several conditions on the variables represented by the set of invariants \( I \). Finally, it is worth noting that each machine contains an initialization event that is executed as the first event in the machine. It assigns different values to the machine variables in order to define the desired initial state (e.g., the number of received requests for a specific content is initialized to 0, or the cached contents set is initialized with the empty set).

To better explain how EBMs work, we will present in detail a simple load balancer intent example. The application developer wants to distribute the load of requests for a video namespace between two producers using the round-robin algorithm. The following intent is then uttered: “Distribute received requests for Video between P1 and P2 using the RoundRobin algorithm” and the following slot values are extracted: Video, P1, P2, and RoundRobin. These slot values are then passed to the abstract EBM template shown in Fig. 8: they serve to initialize the \( \text{slotLoadBalancedNamespace} \), \( \text{slotProducer1} \), \( \text{slotProducer2} \), and \( \text{slotLoadBalancerAlgorithm} \) template parameter variables in the \text{INITIALISATION} event. Fig. 8 shows that EBMs have three main sections: VARIABLES, INVARIANTS, and EVENTS. The VARIABLES section lists all the variables used by the EBM while the INVARIANTS section is initially used to specify the type of these variables. \( \text{ipktContentName} \) and \( \text{ipktDestination} \) are examples of packet variables, while this EBM contains no flow or abstract variables. Every event contains a guards section introduced by the \( \text{WHERE} \) keyword that contains several conditions on the variables, as well as an actions section introduced by the \( \text{THEN} \) keyword where variables are modified. There are two main events for the Ipkts and Dpkts:

### Table 3. Intent to EBM mapping.

| Intent Templates | Abstract EBM templates |
|------------------|------------------------|
| Template name \( n \) and slot sequence \( s_1 \) | Machine \( \mathcal{M}_1 \) |
| Predetermined slot types (e.g., Producers and Namespaces) | Sets in \( S \) in the network context \( C \) |
| Slot labels \( I_{s_2} \subseteq C \times \mathcal{M}_1 \) | Template parameter variables in \( V \) in \( \mathcal{M}_1 \) |
| Possible slot values | Constants in \( C \) of the context \( C \) |
| Semantic of the intent behavior | Constraints represented by invariants \( I \) of \( \mathcal{M}_1 \) |
| Intent behavior implementation | Concrete EBM \( \mathcal{M}_1, I \) logic (events \( E \) acting on variables \( V \)) |

VOLUME 10, 2022
a receive event that initializes the packet variables and a deliver event that allows the receive event to be triggered again. The receive event has an event parameters section introduced by the ANY keyword to represent the possible initialization values of packet variables constrained by a guard condition. For instance, the event parameter contentName of the receiveIpkt event, alongside the guard “contentName ∈ Namespaces”, specify that the Ipkt content name header field can be any namespace from the Namespaces set defined in the context (cf. Fig 7). Finally, there are three events that process Ipkts with the following behavior: if the Ipkt content name is the same as the load-balanced namespace specified in the intent, then the packet is either forwarded to the first or second producers; otherwise, no action is done. As a result, the abstract EBM only describes the details of the namespace check and the packet forwarding, while the exact implementation of the load balancing algorithm is left for the refinement process.

It is worth noting here an essential capability of Event-B that comes from the expressiveness of invariants. While several invariants specify the type of variables, other invariants are used to put constraints on the values of variables. For example, inv8 in Fig. 8 imposes that Ipkts requesting content in the slotLoadBalancedNamespace can only be forwarded either to slotProducer1 or slotProducer2. This constraint corresponds to one part of the semantic of the load balancer intent. Hence, invariants can also be used to represent the expected outcomes of an intent behavior using constraints on variables. Examples of constraints that can be represented as invariants are: the currently served request must belong to the set of authorized contents, the requesting user location must be within a certain geographical area, or the number of responses should not exceed the number of requests in a pull delivery pattern. All the events of the EBM are then checked using the Event-B proof engine (cf. Fig. 4) to make sure they do not violate the constraints set by invariants. As a result, both the invariants and the proof engine result in the “correct by construction” feature of Event-B.

Abstract EBMs are refined to gradually have additional implementation details until the intent behavior is completely specified. In Event-B, a refinement extends an initial EBM by adding new variables, invariants, and events [13]. Events of the abstract EBM can also be refined by adding new guards and actions, with the restriction that the refined event results in exactly the same outcome on the variables of the abstract EBM. This restriction ensures that refined versions of an event may not violate the invariants of the abstract EBM. In other words, refinements are syntactical extensions of an EBM that preserve the invariants. Fig. 9 shows the concrete machine resulting from the refinement of the abstract load balancer machine of Fig. 8 when the round-robin algorithm is used. The currentPosition, numIpktsP1 and numIpktsP2 flow variables are added alongside three invariants that impose the round-robin scheduling constraint (inv4, inv5, and inv6). The processIpktToP1 and processIpktToP2 events are then refined accordingly by adding new guards and actions. In our architecture, we use the refinement patterns concept introduced by Iliasov et al. [44]. Refinement patterns allow us to automate the implementation of refinements by formally specifying every EBM syntactical modification that is part of a refinement. Refinement patterns also have applicability conditions that allow them to be triggered when needed. For instance, the refinement that led to the concrete machine of
TABLE 4. EBM to MAT mapping rules.

| Concrete EBM Component | MAT Component                          |
|------------------------|----------------------------------------|
| Constants in the context C (e.g., content producer) | Switch-specific constants (e.g., output port leading to the producer) |
| Packet variables in V  | Packet header fields, metadata, or special function calls (e.g., execute a meter) |
| Flow variables in V    | Stateful fields (e.g., P4 registers)   |
| Processing step value  | MAT table                              |
| Event guards G_e       | MAT matching key and rules             |
| Event actions A_e      | MAT actions                            |

These consistency checks are based on the fact that every EBM has invariants that specify the expected outcome of the corresponding intent behavior. Consequently, we can check that two EBMs do not conflict with each other by validating the events of the first EBM against the invariants of the second EBM and vice versa. In order to perform these consistency checks, the two EBMs have to be composed to create a combined EBM containing the invariants and events of both machines. The details of EBMs composition are outside the scope of this paper. However, several efficient schemes exist in the literature [45]. The creation of the combined EBM results in the generation of several invariant preservation proof obligations. The Event-B proof engine then examines these proof obligations that require that all events preserve the invariants. Automated tools like Rodin [14] can automatically process most if not all proof obligations; any remaining ones may be proved manually. If a proof obligation cannot be proved, it means that the two intents, or their implementations, are conflicting. The new intent is rejected, and the user who submitted the intent is notified. Once the concrete EBM is validated, it is converted to a data-plane configuration as follows.

The NDN data-plane contains the FIB and PIT tables used to forward the Ipkts and Dpkts, as well as the CS used to cache already served packets. Additionally, we assume that the data-plane contains programmable MATs as part of both the Ipkt and Dpkt pipelines. Examples of implementations of these MATs include our proposed ENDN architecture that uses P4 functions [9], as well as traditional NDN forwarding strategies [12]. A MAT can be used to select custom forwarding actions based on values derived from packet header fields, metadata, or measured statistics. The possible actions include forwarding the packet to one or more network ports, dropping it, sending it to the CS, notifying the controller, modifying header fields, as well as storing a custom state in the switch.

The MAT execution structure can be modeled as a collection of conditional rules of the form if (condition on fields) then do action. The MAT execution structure thus closely resembles the event execution model of EBMs. Hence, we can map EBMs to MATs by following the rules in Table 4.

We can classify the EBM components into four categories: events, variables, context constants, and non-mappable components (e.g., invariants). Events are directly mapped to MAT rules: event guards are mapped to rule conditions, while event actions are mapped to rule actions. Only packet and flow variables can be mapped to a MAT component. Abstract variables are processed by the different refinement patterns.
and are thus not allowed at the level of the concrete machine, while slot parameter variables are considered as constants. Packet variables are standard and have special mapping rules to MAT fields: they are mapped to packet header fields (e.g., content name as shown in Fig. 1), function calls (e.g., execute a meter), or metadata fields (e.g., source and destination ports). Flow variables are usually custom and are mapped to stateful variables in the MAT (e.g., P4 registers). Finally, the context constants are translated to local values for the switch (e.g., a producer is mapped to an output port number and a forwarding hint value). It is worth noting that we can also have special flow variables in EBMs. These can be used to specify some requirements on the FIB and PIT rules (e.g., the FIB routes need to be computed using the shortest path algorithm).

Fig. 10 shows an example of a P4 code corresponding to the round robin load balancer concrete EBM of Fig. 9. The different Event-B components are mapped to the corresponding P4 structures: flow variables become registers (in blue in the code), packet variables become metadata fields or function calls (in green in the code), and context constants become define statements (in red in the code). In the concrete EBM, a special variable called processingStepIpkt allows the events to be organized as possible alternative in a specific processing step of Ipkt. For example, in Fig. 9, the receiveIpkt event corresponds to the processing step 0, then the processIpktToP1, processIpktToP2, and processIpktOtherNamespace can happen at the processing step 1, finally the deliverIpkt event happens during processing step 2. Events that are on the same processing step are mutually exclusive, and thus correspond to different match-action rules in a single MAT. Every processing step thus results in the creation of a new P4 table (e.g., processingStepIpkt1 table in Fig. 10), except for the processing steps of the receive and deliver events. The actions of the events are then mapped to P4 actions accessible from their associated processing step table. Finally, the event guards become entries in the corresponding P4 table. The resulting P4 code can then be installed in the data-plane.

VI. PERFORMANCE EVALUATION
This section demonstrates the advantages of our proposed I2DN architecture. More precisely, declarative goals are expressed as intents, and then translated into data-plane configurations. We then measure the performance gains achieved by these intents when compared to the performance of a traditional NDN configured with shortest path routes and best route strategies [12]. Our experiments employ the Abilene topology [49] built using ENDN switches [9] within the ndnSIM simulator [50]. The ENDN switches are used because they allow our intents to be implemented in the data-plane as P4 functions.

A. TEST SCENARIO
Fig. 11 shows the Abilene topology used in our simulation. All links have a rate of 1Mbps and introduce a propagation delay based on the geographical distance between the cities. We consider a content delivery application with content geo-gating requirements where access to contents is restricted based on the geographical region of the users. More precisely, users from cities on the east coast of the United States (blue nodes in Fig. 11) can only access content specific to their region, and similarly for users from west coast cities (green nodes in Fig. 11). Denver and Indianapolis are regional producers that cache the content of their region, and Kansas City is a national producer that can serve requests...
from both regions while ensuring the geo-gating restrictions using an application-level logic. To configure the network, the application developer initially defines three intents (words in italic correspond to slot values, and the application namespace is /MyApp):

- **I1**: Indianapolis can only serve requests for /MyApp content coming from the east coast.
- **I2**: Denver can only serve requests for /MyApp content coming from the west coast.
- **I3**: Kansas City can serve all requests for /MyApp content.

Additionally, the application developer would like to limit the content requests served by regional producers by automatically offloading any excess requests towards Kansas City. This results in two additional intents:

- **I4**: Limit the /MyApp content requests served by Indianapolis to 100 requests/s and offload any excess requests to Kansas City.
- **I5**: Limit the /MyApp content requests served by Denver to 100 requests/s and offload any excess requests to Kansas City.

We also consider a second application that requires content from the east coast requested by users in the west coast to be delivered with the lowest delay. The content of this application is urgent, so the application developer agreed with the network providers to have the application traffic forwarded with a higher priority. Additionally, the application developer requests proactive caching of the contents in the west coast when the number of requests reaches a certain threshold. In this case, the reception of a new request triggers a secondary request initiated by the P4 code to retrieve other available contents from the east coast to cache them locally. As a result, the application developer selects two intents:

- **I6**: Serve /UrgentContent traffic with high priority.
- **I7**: Proactively cache /UrgentContent contents in the east coast if the number of requests reaches 20 requests per day.

Finally, the network operator selects a strategy intent that locally avoids congestion in the network by always providing two alternative paths to any destination in every switch. The shortest path is used unless the link utilization reaches 90%. In that case, an alternative path is used. This results in the following intent:

- **I8**: Avoid congestion in the network by keeping the link utilization below 90%.

Intents **I1** and **I2** correspond to the same intent template with different slot values (and similarly for intents **I4** and **I5**). The different intents are then processed by our architecture and result in several P4 functions that are placed in the switches as follows:

- The P4 functions corresponding to intents **I1** and **I2** are placed in the east coast and west coast nodes respectively (i.e., the green and blue nodes in Fig. 11). These P4 functions add a forwarding hint towards Indianapolis or Denver to the /MyApp Ipks originating from the east or west coasts respectively.
- **I3** is translated to a P4 function placed in Kansas City that automatically sends Ipks to the central producer even if a forwarding hint to Denver or Indianapolis is present.
- **I4** and **I5** are mapped to a rate-limiting P4 function placed in Denver and Indianapolis. It measures the rate of /MyApp requests and offloads any traffic over 100 requests/s to Kansas City.
- **I6** is implemented as a P4 function that requires all /UrgentContent packets to be processed with a high queue priority. This P4 function is installed in all the switches along the path followed by /UrgentContent packets.
- **I7** is translated to a P4 function placed in Denver that proactively caches the /UrgentContent in the local CS.
- The P4 function generated by **I8** is placed in all the switches. It processes all Ipks containing forwarding hints towards specific destinations (e.g., Denver or Indianapolis in our scenario) by sending them to a secondary path in case of congestion. The algorithm also makes sure to check the source port from which packets are received to avoid creating forwarding loops by sending the packet back through the face from where it was received.

At $t = 0s$, consumers from every city of the east and west coasts (i.e., the green and blue nodes in Fig. 11) start requesting /MyApp content at a rate proportional to the size of their population. From $t = 100s$ to $t = 150s$, there is a rush period where additional traffic is added, resulting in congestion on the east coast. Finally, the /UrgentContent located in Atlanta is requested by a consumer in Seattle at a slow exponentially distributed rate with a mean of 1 request/s during the entire simulation time. The RTT (including transmission, propagation, and queuing delays), packet loss rate, and received Dpkt throughput are measured for the /MyApp traffic originating from every city as well as for the /UrgentContent traffic. We then compare the performance of an NDN network configured using the intents described above against that of a standard NDN network with no intents. The latter forwards all /MyApp requests to Kansas City using the shortest path as geo-gating can only be guaranteed by the central producer.

**B. EXPERIMENTAL RESULTS**

Fig. 12 shows the measured RTT for the /MyApp traffic originating from Los Angeles, Houston, and New York.
The effects of satisfying intents $I_1$ and $I_2$ are visible in Figs. 12a and 12b: the RTT increases by around 30ms when no intents are used because the packets are served by the central producer in Kansas City instead of the regional producers. This additional delay is consistent with the propagation delay of 10ms between the regional producers and Kansas City added twice to the transmission delay of a 1KB Dpkt over the 1Mbps links. On the other hand, $I_3$ allows the requests originating from Houston to be processed directly by the central producer, which is closer than the regional producers. During the rush period, the traffic is increased, which causes the Indianapolis rate-limiting threshold defined by $I_4$ to be reached. The excess traffic is thus offloaded to Kansas City which causes the RTT of the New York traffic to increase slightly in Fig. 12a.

Fig. 12d clearly shows the effects of intents $I_6$ and $I_7$. At around $t = 20s$, the Denver switch proactively caches the /UrgentContent Dpkts which causes a significant decrease of the RTT. Additionally, the delay remains unchanged during the rush period when intents are used as the /UrgentContent traffic is treated with a high queue priority.

The congestion also causes an increase in packet loss and a decrease in received throughput as shown in Figs. 13a, 13d, 14a, and 14d. On the other hand, the P4 function that was generated from $I_8$ has successfully avoided the congestion. Hence, there is no degradation of performance when intents are used.

Our proposed architecture has allowed the network to adapt to the needs of the application and network operators while improving network manageability and configuration through intents. This, in turn, resulted in a better network performance compared to traditional non-intent-based networks.

C. COMPUTATIONAL COST

Finally, we analyze the computational cost that is introduced by the intents on both the control- and data-planes.

At the control-plane, our architecture processes intents asynchronously from the data-plane operations: data-plane configurations are generated once when an intent is processed but are not modified later during packet processing. More precisely, every intent is completely translated into an autonomous data-plane configuration/program that does not interact with the control-plane. Hence, the communication overhead between the control- and data-planes takes place once. The operator can limit data-plane updates to batch processes. The main control-plane cost is incurred during the installation of a new configuration in the data-plane. However, several programmable data-plane architectures allow
FIGURE 13. Measured packet loss for the /MyApp traffic coming from different cities and for the /UrgentContent traffic.

FIGURE 14. Received Dpkt throughput measured for the /MyApp traffic coming from different cities and for the /UrgentContent traffic.
fast runtime reconfigurability of P4 programs which makes the impact of data-plane reconfigurations minimal on the switch operation [9].

At the data-plane level, P4 programs introduce a processing delay dependent on the switch hardware or software implementation [51]. Several high-performance P4 switch implementations were proposed in the literature to significantly reduce this processing delay, especially using FPGAs [52] or GPUs [53]. It is worth noting, however, that most high-performance P4 switches are limited in the number of P4 programs that can be executed in parallel (e.g., P4VBox can execute up to 13 P4 programs in parallel [54]). Hence, the main data-plane cost overhead can be characterized by the number of P4 functions that are needed in the network for a specific set of intents. In our test scenario, we notice that some intents are mapped to a single P4 function (e.g., I3 or I4), while other intents are implemented as a P4 function placed in every switch (e.g., I6 or I8). It is worth noting though that several intents correspond to the same intent template with different slot values. These intents can thus be shared at the data-plane level by calling the same P4 function using different parameters. The number of P4 functions at the data-plane level can thus be reduced using P4-function sharing. In previous work [9], the authors have discussed the trade-off between scalability and intent customizability and performance that depends on the available MAT resources at the data-plane level. This trade-off is decided by the network operator and embedded in a control-plane logic at the level of the EBM analyzer. The details of this logic consist in solving constrained optimization problems and are outside the scope of this paper.

VII. OPEN ISSUES AND FUTURE RESEARCH WORK
This section discusses several assumptions and limitations of our proposed work and highlights future research opportunities.

- **Intent model:** The proposed intent model takes a major step forward towards representing intents that can capture the operator and developer goals in a much more declarative way compared to the traditional event-condition-action models [3]. However, the model relies on predefined classes of intents where users must utter one of the predefined intents. This model represents a closed-world model. An open-world intent model can accept and identify unknown, not previously seen, intents from the users. In the Literature, several open-world and multiple intents models have been developed in other contexts, such as chatbots [42], but remain a challenge for IDN.

- **Single vs. multiple network domains** In the proposed work, we considered a single subnetwork with a single control domain. Extending the proposed work to multiple independent domains that necessitate the collaboration and orchestration between several controllers is left as future work.

- **Learning and run-time adaptation:** Thus far, our work has focused on the mapping of user intents to PDP configurations while assuring conflict resolution and validation before they are installed. We believe that producing an efficient intent model and intent to data-plane translation methodology represents a first step towards realizing self-configuring and healing IDN. Hence, the challenge of monitoring and analyzing the network behavior and adapting it at run-time remains a future work.

- **Trust and security:** Allowing application developers to configure the network data-plane indirectly through intents introduces additional trust and security issues.

VIII. CONCLUSION AND FUTURE WORK
This paper proposed a novel architecture to capture high-level named-data network (NDN) service intents and translate them into data-plane configurations. Our architecture employs the Event-B modeling and refinement concepts to represent high-level intents using abstract Event-B Machines (EBMs) and then refine them to machines that can be used to configure the data-plane. We have provided a detailed description of the modeling and mapping steps for translating intents to EBMs and refining these machines. Finally, we showed how these produced EBMs could be translated to instructions on the data-plane match action tables. Experimental evaluation results demonstrate the feasibility and efficiency of the various functionalities of the architecture. Currently, we are investigating the feasibility of employing deep learning to replace some of the statically defined mapping rules.

REFERENCES
[1] L. Zhang, A. Afanasyev, J. Burke, V. Jacobson, K. Claffy, P. Crowley, C. Papadopoulos, L. Wang, and B. Zhang, “Named data networking,” *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 66–73, Jul. 2014.
[2] Z. Li, Y. Xu, B. Zhang, L. Yan, and K. Liu, “Packet forwarding in named data networking requirements and survey of solutions,” *IEEE Commun. Surveys Tuts.*, vol. 21, no. 2, pp. 1950–1987, 2nd Quart, 2019.
[3] E. Zeydan and Y. Turk, “Recent advances in intent-based networking: A survey,” in *Proc. IEEE 91st Veh. Technol. Conf. (VTC-Spring)*, May 2020, pp. 1–5.
[4] L. Pang, C. Yang, D. Chen, Y. Song, and M. Guizani, “A survey on intent-driven networks,” *IEEE Access*, vol. 8, pp. 22862–22873, 2020.
[5] R. Ullah, M. A. U. Rehman, and B.-S. Kim, “Design and implementation of an open source framework and prototype for named data networking-based edge cloud computing system,” *IEEE Access*, vol. 7, pp. 57741–57759, 2019.
[6] O. Karrakchou, N. Samaan, and A. Karmouch, “EP4: An application-aware network architecture with a customizable data plane,” in *Proc. IEEE 22nd Int. Conf. High Perform. Switching Routing (HPSR)*, Jun. 2021, pp. 1–6.
[7] N. Anerousis, P. Chemouil, A. A. Lazar, N. Mihai, and S. B. Weinstein, “The origin and evolution of open programmable networks and SDN,” *IEEE Commun. Surveys Tuts.*, vol. 23, no. 3, pp. 1956–1971, 3rd Quart, 2021.
[8] P. Bossart, G. Varghese, D. Walker, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, and A. Vahdat, “P4: Programming protocol-independent packet processors,” *SIGCOMM Comput. Commun. Rev.*, vol. 44, pp. 87–95, Jul. 2014.
L. Chen, O. Havel, A. Olariu, P. Martinez-Julia, J. Nobre, and M. Paliwal, D. Shrimankar, and O. Tembhurne, “Controllers in SDN: A X. Li, N. Samaan, and A. Karmouch, “An automated VNF manager based Q. Sun, W. Wendong, Y. Hu, X. Que, and G. Xiangyang, “SDN-based A. Kalghoum and S. M. Gammar, “Towards new information cen- K. Thar, N. H. Tran, S. Ulah, T. Z. Oo, and C. S. Hong, “Novel cooperative and fully-distributed congestion control mechanism for content centric networking.” IEEE Access, vol. 5, pp. 27691–27706, 2017. M. A. Naeem, M. A. U. Rehman, R.ullah, and B.-S. Kim, “A comparative performance analysis of popularity-based caching strategies in named data networking,” IEEE Access, vol. 8, pp. 50057–50077, 2020. K. Thar, N. H. Tran, S. Ulah, T. Z. Oo, and C. S. Hong, “Online caching and cooperative forwarding in information centric networking.” IEEE Access, vol. 6, pp. 59679–59694, 2018. A. Kalghoum and S. M. Gammar, “Towards new information centric networking strategy based on software defined networking:” in Proc. IEEE Wireless Commun. Netw. Conf. (WCNC), Mar. 2017, pp. 1–6. Q.-Y. Zhang, X.-W. Wang, M. Huang, K.-Q. Li, and S. K. Das, “Software defined networking meets information centric networking: A survey,” IEEE Access, vol. 6, pp. 39547–39563, 2018. E. Aubry, T. Silverston, and I. Chrisimen, “Implementation and evaluation of a controller-based forwarding scheme for NDN.” in Proc. IEEE 31st Int. Conf. Adv. Inf. Netw. Appl. (AINA), Mar. 2017, pp. 144–151. Q. Sun, W. Wendong, Y. Hu, X. Que, and G. Xiangyang, “SDN-based autonomous CCN traffic management,” in Proc. IEEE Globecom Workshops (GC Wkshps), Dec. 2014, pp. 183–187. R. Jmal and L. C. Fourati, “An OpenFlow architecture for managing content-centric network (OFAM-CCN) based on popularity caching strategy,” Comput. Stand. Interface, vol. 51, pp. 22–29, Mar. 2017. P. Zuraniwski, N. van Adrichem, D. Ravestein, W. Jintema, C. Papadopoulos, and C. Fan, “Facilitating ICN deployment with an extended openflow protocol,” in Proc. 4th ACM Conf. Inf.-Centric Netw., Sep. 2017, pp. 123–133. L. Chen, O. Havel, A. Olariu, P. Martinez-Julia, J. Nobre, and D. Lopez. Intent Classification. Accessed: Nov. 23, 2021. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-irtf-nmrg-ibn-intent-classification-05 Y. Wang, R. Forbes, U. Elzur, J. Strassner, A. Gamelas, H. Wang, S. Liu, L. Pesando, X. Yuan, and S. Cai, “From design to practice: ETSI ENI reference architecture and instantiation for network management and orchestra- tion using artificial intelligence.” IEEE Commun. Standards Mag., vol. 4, no. 3, pp. 38–45, Sep. 2020. S. Perez-Soler, S. Suarez-Puerta, E. Guerra, and J. de Lara, “Choosing a chatbot development tool.” IEEE Softw., vol. 38, no. 4, pp. 94–103, Jul. 2021. X. Li, N. Samaan, and A. Karmouch, “An automated VNF manager based on parameterized action MDP and reinforcement learning.” in Proc. IEEE Int. Conf. Commun., Jun. 2021, pp. 1–6. M. Paliwal, D. Shrimankar, and O. Tembhome, “Controllers in SDN: A review report.” IEEE Access, vol. 6, pp. 36256–36270, 2018. 31 Y. Han, J. Li, D. Huang, J.-H. Yoo, and J. W.-K. Hong, “An intent-based network virtualization platform for SDN,” in Proc. 12th Int. Conf. Netw. Service Manage. (CNSM), Oct. 2016, pp. 353–358. Y. Takuma and Y. Okabe, “Reactive configuration updating for intent- based networking.” in Proc. Int. Conf. Inf. Netw. (ICION), 2017, pp. 97–102. T. A. Khan, A. Mohammad, K. Abbas, and W.-C. Song, “Intent-based networking platform: An automated approach for policy and configuration of next-generation networks,” in Proc. 36th Annu. ACM Symp. Appl. Comput. (SAC), Mar. 2021, pp. 1921–1930. V. Heorhii, S. Chandrasekaran, M. K. Reiter, and V. Sekar, “Intent- driven composition of resource-management SDN applications,” in Proc. 14th Int. Conf. Emerg. Netw. Exp. Technol., Dec. 2018, pp. 86–97. M. Riffati and F. Kuipers, “P4I/O: Intent-based networking with P4,” in Proc. IEEE Conf. Netw. Softwarization (NetSoft), Jun. 2019, pp. 438–443. A. Singh, G. S. Aujla, and R. S. Bal, “Intent-based network for data dis- semination in software-defined vehicular edge computing,” IEEE Trans. Intell. Transp. Syst., vol. 22, no. 8, pp. 5310–5318, Aug. 2021. T. Kollar, D. Berry, L. Stuart, K. Owczarzak, T. Chung, L. Mathias, M. Kayser, B. Snow, and N. Matsoukas, “The Alexa meaning repre- sentation language.” in Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics, Hum. Lang. Technol.,” (Industry Papers), vol. 3, 2018, pp. 179–184. A. de Barcelos Silva, M. M. Gomes, C. A. da Costa, R. da Rosa Righi, J. L. V Barbosa, G. Pessin, G. D. Doncker, and G. Federizzi, “Intelligent personal assistants: A systematic literature review,” Expert Syst. Appl., vol. 147, Jun. 2020, Art. no. 113193. Z. Zhang, Z. Zhang, H. Chen, and Z. Zhang, “A joint learning framework with BERT for spoken language understanding,” IEEE Access, vol. 7, pp. 168849–168858, 2019. K. Abbas, T. A. Khan, M. Afaq, and W.-C. Song, “Network slice life- cycle management for 5G mobile networks: An intent-based networking approach,” IEEE Access, vol. 9, pp. 80128–80146, 2021. G. Bianchi, M. Bonola, A. Capone, and C. Cassocne, “Openstate: Pro- gramming platform-independent stateful openflow applications inside the switch,” ACM SIGCOMM Comput. Commun. Rev., vol. 44, no. 2, pp. 44–51, 2014. H. E. Z. Zhan and M. Song, “Table-to-dialog: Building dialog assis- tants to chat with people on behalf of you,” IEEE Access, vol. 8, pp. 102313–102320, 2020. S. Chen and S. Yu, “WAIS: Word attention for joint intent detec- tion and slot filling,” in Proc. AAAI Conf Artif. Intell., vol. 33, 2019, pp. 9927–9928. A. Biaos, E. Troubitsyna, L. Laibinis, and A. Romanovsky, “Patterns for refinement automation,” in Formal Methods for Components and Objects (Lecture Notes in Computer Science), S. de Boer Frank, M. Marcello, S. H. Bonsange, and M. Leuschel, Eds. Berlin: Springer, 2010, pp. 70–88. R. Silva and M. Butler, “Shared event composition/decomposition in event-B,” in Formal Methods for Components and Objects (Lecture Notes in Computer Science), B. K. Aichernig, S. de Boer Frank, and M. Bonsange, Eds. Berlin: Germany: Springer, 2012, pp. 122–141. A. L. R Madureira, F. R. C Araujo, G. B. Araujo, and L. N. Sampaio, “NDN fabric: Where the software-defined networking meets the content- centric model,” IEEE Trans. Netw. Service Manage., vol. 18, no. 1, pp. 374–387, Mar. 2021. J. C. Fillmore and C. Baker, A Frames Approach to Semantic Analysis. London, U.K.: Oxford Univ. Press, Dec. 2009. O. Sattarow, “Natural language processing with DeepPavlov library and additional semantic features,” in Artificial Intelligence (Lecture Notes in Computer Science), G. S. Osipov, A. I. Panov, and K. S. Yakovlev, Eds. Cham, Switzerland: Springer, 2019, pp. 146–159. Abilene Core Topology | University IT Available: https://uit.stanford.edu/service/network/internet2/abilene Chunks. A Frames Approach to Semantic Analysis. London, U.K.: Oxford Univ. Press, Dec. 2009. O. Sattarow, “Natural language processing with DeepPavlov library and additional semantic features,” in Artificial Intelligence (Lecture Notes in Computer Science), G. S. Osipov, A. I. Panov, and K. S. Yakovlev, Eds. Cham, Switzerland: Springer, 2019, pp. 146–159. Abilene Core Topology | University IT Available: https://uit.stanford.edu/service/network/internet2/abilene Chunks. A Frames Approach to Semantic Analysis. London, U.K.: Oxford Univ. Press, Dec. 2009. O. Sattarow, “Natural language processing with DeepPavlov library and additional semantic features,” in Artificial Intelligence (Lecture Notes in Computer Science), G. S. Osipov, A. I. Panov, and K. S. Yakovlev, Eds. Cham, Switzerland: Springer, 2019, pp. 146–159. Abilene Core Topology | University IT Available: https://uit.stanford.edu/service/network/internet2/abilene Chunks. A Frames Approach to Semantic Analysis. London, U.K.: Oxford Univ. Press, Dec. 2009.
OUASSIM KARRAKCHOU (Graduate Student Member, IEEE) received the master’s degree in telecommunications engineering from the Institut National des Sciences Appliquées (INSA), Lyon, France, in 2014. He is currently pursuing the Ph.D. degree with the School of Electrical and Computer Engineering, University of Ottawa. He has worked as a Technical Consultant for a leading financial software editor in France, from 2014 to 2017. His research interests include future internet architectures, information-centric networks, software-defined networks, and cloud.

NANCY SAMAAN (Member, IEEE) received the B.Sc. and M.Sc. degrees from the Department of Computer Science, Alexandria University, Egypt, and the Ph.D. degree in computer science from the University of Ottawa, Canada, in 2007. She is currently a Professor with the School of Electrical Engineering and Computer Science, University of Ottawa. Her current research interests include network resource management, wireless communications, quality-of-service issues, and autonomic communications. In 2008, she received the Natural Sciences and Engineering Research Council of Canada University Faculty Award.

AHMED KARMOUCH (Member, IEEE) received the M.S. and Ph.D. degrees in computer science from the University of Paul Sabatier, Toulouse, France, in 1976 and 1979, respectively. He was an Industrial Research Chair at the Ottawa Carleton Research Institute and the Natural Sciences and Engineering Research Council. He has been the Director of the Ottawa Carleton Institute for Electrical and Computer Engineering. He is currently a Professor at the School of Electrical Engineering and Computer Science, University of Ottawa. He is involved in several projects with industry and government laboratories in Canada and Europe. His current research interests include mobile computing, autonomic overlay networks, software defined networks, cloud computing, and network virtualization. He has organized several conferences and workshops, edited several books, and served as a Guest Editor for IEEE Communications Magazine, Computer Communications, and others.

* * *