Octant, la vérification réseau simplifiée

Pierre-Léo Bégay, Pierre Crégut, Jean-François Monin

Octant,
la vérification réseau simplifiée

Pierre-Léo Bégay[1], Pierre Crégut[2], and Jean-François Monin[3]

[1,2]Orange Labs
[1,3]Verimag
[1]pierreleo.begay@orange.com, [2]pierre.cregut@orange.com,
[3]jean-francois.monin@univ-grenoble-alpes.fr

Résumé

Octant est un outil simplifiant la vérification de propriétés de connectivité dans des réseaux SDN. Il se base sur l'implémentation Z3 d'un Datalog orienté réseau et l'étend avec des optimisations globales transformant des programmes génériques, simples à développer et analyser mais ne passant pas à l'échelle, en des versions spécialisées efficaces.

1 Introduction

Datalog est un langage déclaratif dérivé de Prolog, adapté aux applications pour la manipulation de données et dont les programmes sont simples à écrire et comprendre. Il a récemment été utilisé dans la création d'un outil de vérification de propriétés de connectivité (accessibilité, isolation, double attachement, redondance des chemins, etc.) de déploiements réseaux, appelé NoD [2]. NoD passe à l'échelle sur des problèmes de taille industrielle, au prix d'optimisations manuelles et complexes transformant des programmes génériques en des versions bien plus longues et souvent illisibles.

Nous présentons Octant, un outil basé sur NoD, utilisant des optimisations automatiques, formalisées et vérifiées dans l'assistant de preuve Coq. La section 2 présente les bases de la modélisation de réseaux avec Datalog, puis la section 3 introduit nos optimisations, et enfin on présente en 4 différents détails de la modélisation.

2 Modélisation réseau avec Datalog

2.1 Besoins des modèles réseaux

La virtualisation réseau utilise des outils logiciels pour créer, sur les mêmes ressources matérielles, plusieurs réseaux isolés les uns des autres.

Implémenter des éléments réseaux via des couches logicielles permet d'obtenir des architectures plus dynamiques et adaptables, mais également plus complexes à comprendre et vérifier, et donc plus fragiles.

Un *virtual network manager*, comme Neutron dans OpenStack, connecte les différentes ressources de calcul et de stockage d'une infrastructure cloud, en utilisant un modèle central pouvant être étendu par des *service plugins* (pare-feux, répartition de charge, chaînage de fonctions de service, interconnexion de data-centers, etc). Ces infrastructures sont ensuite utilisées par les opérateurs télécom pour déployer des machines virtuelles, ou conteneurs, implémentant des fonctions réseaux virtuelles opérant sur le trafic.

Les réseaux privés sont implémentés comme des overlays sur l'infrastructure physique. Plusieurs mécanismes d'overlay et de routage peuvent être choisis, altérant potentiellement le comportement de l'abstraction obtenue. Nous avons donc besoin d'un langage pour décrire le comportement de tous les éléments de routage sur toutes les couches, ainsi que la topologie, et vérifier des propriétés de paquets abstraits sur le modèle produit. Nous avons choisi Datalog pour son expressivité et sa simplicité.

Notre outil Octant étant destiné aux équipes chargées du développement des infrastructures, les modèles doivent être faciles à écrire, comprendre et adapter aux changements dans la sémantique des éléments virtuels. L'enjeu est de passer à l'échelle sans sacrifier la simplicité de ses modèles.

Octant interroge les bases de données des services d'OpenStack ou de Skydive[1], un analyseur de réseau produisant un modèle de la topologie d'une infrastructure virtuelle. Les propriétés à vérifier sont décrites dans un dialecte typé de Datalog. Octant applique des transformations globales sur ces programmes avant de les traduire dans NoD.

2.2 Datalog

Datalog est un langage logique déclaratif, dont les programmes sont des clauses de Horn, parfois appelées règles. Les atomes des clauses contiennent des variables ou des constantes mais pas de symbole de fonction. Alternativement, on peut voir Datalog comme une extension du calcul relationnel équipé de la récursion, ce qui permet de l'utiliser dans les langages de requêtes de bases de données.

Datalog distingue les prédicats extensionnels, dont les faits forment la base de données extensionnelle (EDB), des prédicats intentionnels, dont les faits sont déduits en itérant les clauses en partant de l'EDB.

On peut étendre Datalog avec la négation, auquel cas le programme doit pouvoir être stratifié de sorte que négation et récursion ne se croisent pas, ainsi qu'avec des prédicats primitifs sur des types standards, ici les opérations booléennes sur les vecteurs de bits et des comparaisons arithmétiques.

2.3 Modèles réseaux en Datalog

2.3.1 Accessibilité en général

On suppose une relation `link(from:id, to:id)` dans l'EDB, représentant un lien entre deux éléments réseaux dénotés par des identifiants. En première approximation, l'accessibilité (ici d'un élément plutôt que d'une adresse, pour simplifier les règles) est calculée à l'aide de la récursion de Datalog comme la clôture transitive de la relation `link`. Les arguments des prédicats extensionnels sont nommés explicitement, ceux omis dans l'utilisation d'un atome correspondent à des variables silencieuses.

```
simple_reach(P, P).
simple_reach(P, T) :- simple_reach(P, F), link(from=F, to=T).
```

2.3.2 Basculement

En pratique, la propagation des paquets dans un réseau dépend bien sûr de la topologie, mais aussi des règles des routeurs. Un paquet est concerné par une règle si la conjonction bit à bit (&) de la destination du paquet avec le *mask* de la règle équivaut au préfixe de cette dernière. Ce mécanisme est codé dans la règle suivante, où les routeurs sont représentés par un `id`, tandis que les paquets sont codés par leur destination, de type `ip`.

```
match(IP, F, T, P) :- rule(from=F, mask=M, prefix=R,
    to=T, priority=P), IP & M = R.
```

Plusieurs règles peuvent *matcher* un même paquet, auquel cas celle avec la plus grande priorité est sélectionnée. Le prédicat `exists_better(IP,F,P)` indique l'existence d'une règle `P'` prioritaire sur `P`. L'optimalité d'une règle est donc exprimée à l'aide de la négation `~exists_better`.

```
exists_better(IP, F, P) :- match(IP, F, T, P'), P' > P.
reach(IP, T) :-
   reach(IP,F), match(IP, F, T, P), ~exists_better(IP, F, P).
```

En pratique, les paquets sont représentés par un n-uplet d'au moins 5 éléments (protocole IP, adresses d'origine et destination, port(s), longueur du préfixe...). Cette forme de sélection de règles de routage est omniprésente, en particulier l'utilisation de la négation pour éliminer les paquets déjà traités par des règles à plus haute priorité : cependant il y a de nombreuses variantes, en fonction de la partie étudiée dans l'en-tête ou de la gestion des priorités des règles.

3 De la généricité à l'efficacité

L'exécution de ces exemples par un moteur Datalog standard, cherchant les solutions par énumération, ne passera pas à l'échelle sur des problèmes de taille industrielle. Il faut d'abord une représentation adaptée des en-têtes de paquets. Il est également nécessaire de transformer les programmes pour les rendre plus efficaces avec les nouvelles structures de données.

3.1 Network Oriented Datalog

Datalog utilisant des domaines finis, les programmes terminent toujours. Cependant, sur de très grands domaines, les garanties de terminaison n'ont plus de sens pratique, et des représentations adaptées au contexte sont nécessaires. Lopes et at [2] ont montré avec leur outil Network Oriented Datalog (NoD) que les Différences de Cubes (DoC) forment une représentation adaptée pour les modèles réseaux.

3.2 Spécialisation des prédicats

Une de nos optimisations consiste à introduire de nouveaux prédicats pour partitionner les relations existantes en des versions réduites. Quand un argument d'un prédicat intentionnel est systématiquement une constante dans les règles définissant ledit prédicat, on peut remplacer ce dernier par un ensemble de versions spécialisées. Si par exemple `P` est défini par les règles

```
P(1,Y,Z) :- Q(Y,Z).
P(2,Y,Z) :- R(Z,Z,Y).
```

alors on introduit les prédicats `P_1` et `P_2`, d'arité 2. Les règles précédentes sont remplacées par

```
P_1(Y,Z) :- Q(Y,Z).
P_2(Y,Z) :- R(Z,Z,Y).
```

Pour que les règles contenant un `P` dans leur corps puissent toujours être utilisées, on ajoute les règles

```
P(1,Y,Z) :- P_1(Y,Z).
P(2,Y,Z) :- P_2(Y,Z).
```

La complexité de la plupart des opérations combinant des DoCs n'étant pas linéaire par rapport aux représentations, la réduction des structures améliore significativement l'efficacité. De plus, si la spécialisation n'est pas une idée nouvelle, c'est en combinaison avec l'analyse statique au cœur de notre autre optimisation qu'elle prend tout son intérêt.

3.3 Analyse statique des clauses de Horn

Des prédicats primitifs à plusieurs variables, comme $IP\&M = R$ sont mal gérés par NoD. Notre solution est d'utiliser le fait que, dans la plupart des cas, les valeurs de deux de ces variables (préfixe et masque) sont issues des relations de l'EDB. Le même raisonnement vaut pour les comparaisons sur les priorités : dans l'encodage du *routing* présenté en 2.3.2 par exemple, la clause définissant `exists_better` pourrait être remplacée par plusieurs copies dans lesquelles la variable `R` est instanciée avec les différentes valeurs de `prefix` dans les faits de l'EDB définissant `rule`.

Fournir l'ensemble exact de valeurs possibles de n'importe quelle variable n'est pas un objectif raisonnable, car revenant à exécuter le programme afin de l'optimiser. Cependant, dupliquer une clause générale avec des valeurs n'apparaissant pas dans l'exécution du programme n'en altère pas la sémantique, puisque ces nouvelles clauses auraient des atomes non-satisfiables dans leur corps, empêchant la déduction de faits nouveaux. Nous utilisons une analyse statique qui calcule, de façon efficace, une sur-approximation des valeurs allant de l'EDB aux différentes clauses dans l'exécution.

Cette analyse associe à chaque variable l'origine de ses valeurs sous forme de formule de logique propositionnelle. Les propositions de base sont des couples formés d'un prédicat extensionnel et un index, représentant un ensemble de valeurs dans l'EDB. Pour raisonner sur la structure syntaxique des programmes, on suppose que les règles sont numérotées (à partir de 0), tout comme les atomes dans leur corps (la tête est mise à part) et les arguments de ceux-ci. Par exemple, $\langle f, 1\rangle$ correspond à l'ensemble des valeurs en deuxième position de f dans l'EDB.

L'analyse d'une variable nécessite également celle d'arguments de prédicats. Pour analyser `X0` dans le programme de la figure 1b, on doit pouvoir calculer un sur-ensemble des valeurs que peuvent prendre les premiers arguments de `Q`, `R` et `S`, et en renvoyer l'intersection. Pour ce qui est par exemple de `Q`, on a deux règles permettant d'en déduire des faits, on doit donc renvoyer l'union des valeurs que peuvent prendre les variables `X1` et `X2`. Les branches sont étiquetées avec les index des atomes ou des règles correspondant. Le cas de base est l'analyse d'un argument d'un prédicat extensionnel, auquel cas on renvoie le couple formé du prédicat et de l'index de l'argument. Par exemple, la figure 2 représente l'analyse des variables `X0` et `Y0` de la première règle du programme de la figure 1b.

Datalog gérant la récursion, notre analyse doit le faire aussi afin de ne pas boucler. Nous ajoutons donc simplement en argument la liste des points de programme déjà analysés, et on arrête la branche actuelle quand on est sur un élément déjà traité. L'idée est que, pour trouver une sur-approximation des valeurs que peut prendre une variable, il ne faut pas s'intéresser à sa partie récursive, mais aux autres prédicats qui en contraignent les valeurs. Par exemple, dans le programme de la figure 1a, les faits déduits pour `P` sont

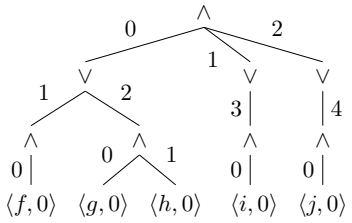
```
P(X0,Y0) :- Q(Y0,X0).
P(X1,Y1) :- f(X1,Y1).
Q(X2,Y2) :- P(X2,Y2).
```

(a) Programme Datalog récursif

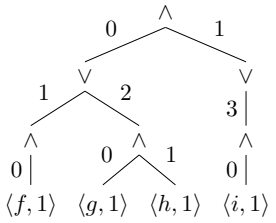
```
P(X0,Y0) :- Q(X0,Y0), R(X0,Y0), S(X0).
Q(X1,Y1) :- f(X1,Z1).
Q(X2,Y2) :- g(X2,Y2,Z2), h(X2,Y2,Z2).
R(X3,Y3) :- i(X3,Y3).
S(X4)    :- j(X4).
```

(b) Programme avec des choix

FIGURE 1 – Exemples de programmes Datalog



(a) Analyse de X0



(b) Analyse de Y0

FIGURE 2 – Deux analyses de variables

ceux de `f` *modulo* permutation *via* `Q`, ce que capture l'analyse.

Chaque arbre nous permet, individuellement, d'extraire un ensemble de valeurs pour la variable correspondante. Cependant, procéder ainsi nous mène à instancier la règle avec le produit cartésien des valeurs extraites, alors que de nombreux n-uplets ne correspondent à rien. Dans la figure 2, les branches $[0-1-0]$ de 2a et $[0-2-0]$ de 2b sont incompatibles, en ce qu'elles supposent que le premier atome de la première règle, dans lequel $X0$ et $Y0$ apparaissent tous les deux, est instancié avec deux règles différentes. On veut donc utiliser les annotations pour superposer les arbres des différentes variables, ce qui transforme par exemple ceux de la figure 2 en la figure 3, où $\top$ indique l'absence de contrainte.

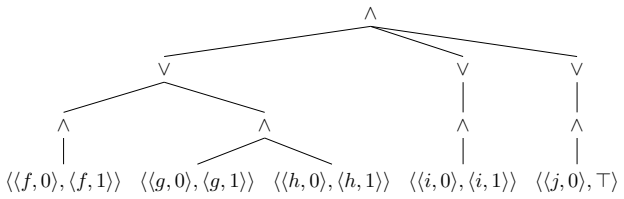


FIGURE 3 – Analyses combinées

Dans les programmes d'Octant, les identifiants d'éléments réseaux sont utilisés comme clefs de jointure explicites dans toutes les relations définissant le trafic. Après la spécialisation des règles, ces identifiants sont généralement eux-mêmes spécialisés, produisant une règle explicite pour chaque identifiant

utilisé et préparant le terrain pour la spécialisation des prédicats.

La spécialisation des prédicats et l'analyse statique individuelle, qui représentent un total de 1200 lignes de code Python, ont été modélisées et validées en Coq. Les règles d'Octant permettent la superposition des analyses décrite plus haut, mais elle ne fonctionne pas dans le cas général (la figure 4 est un contre-exemple). Une caractérisation précise des programmes permettant cette optimisation supplémentaire reste à déterminer.

```
P(X,Y,Z) :- P(Y,X,Z).
P(X,Y,Z) :- Q(X,Y,Z).
```

FIGURE 4 – Récursion non-homogène

4 Modèles réseau en Datalog

Pour vérifier les propriétés d'un déploiement réseau, Octant en aspire la configuration pour remplir son EDB et lui appliquer des programmes Datalog. Les tables des bases relationnelles d'OpenStack ou le graphe (décrit sous forme de structures JSON typées) de Skydive sont traduits en prédicats Datalog. Les programmes sont décrits dans un Datalog typé traduit dans le Datalog orienté réseau de Z3 après application des transformations décrites plus haut. Tous les types de données sont transformés en vecteurs de bits (adresses IP, identifiants et chaînes de caractères opaques, entiers).

4.1 OpenStack

Nous avons décrit en Datalog le modèle de données de Neutron, le gestionnaire de réseau d'OpenStack. Neutron permet de modéliser des réseaux virtuels connectés à des équipements (machines virtuelles ou routeurs) par des ports ayant des adresses IP. Outre le routage entre réseaux adjacents, la modélisation gère les routes additionnelles à la fois au niveau réseau et au niveau routeur, les groupes de sécurité et les règles de pare-feu. L'ensemble du modèle (environ 200 lignes de code) utilise les priorités à différents niveaux (entre règles de routage, règles de pare-feu, ou groupes de sécurité). Des topologies de plusieurs dizaines de serveurs ont été décrites. En utilisant le modèle de Hassel [1], des propriétés simples sur des topologies de plusieurs centaines d'éléments [2] ont été vérifiées.

1. https ://bitbucket.org/peymank/hassel-public
2. http ://web.ist.utl.pt/nuno.lopes/netverif/

4.2 OpenFlow

Nous avons modélisé en Datalog le cœur d'OpenFlow[3], un langage standardisé permettant de décrire les actions de routage sous forme de règles avec priorité. Il est en particulier utilisé par OpenVSwitch[4], un commutateur virtuel très utilisé dans les infrastructures de cloud. Les données de base qui constituent l'EDB sont récupérées par Skydive sur la configuration des commutateurs OpenVSwitch.

Une règle est constituée d'une liste de filtres et d'une liste d'actions. La principale difficulté est la modélisation des listes (structures récursives) en Datalog. Tout constructeur de la liste a un identifiant unique. Le prédicat modélisant chaque élément intègre aussi la structure de la liste (constructeur courant et suivant). Une liste $a(x) :_1 b(y,z) :_2 []_3$ est transformée en $a(1,2,x), b(2,3,y,z), nil(3)$. Chaque règle est codée comme une transformation d'état enchaînant d'abord le filtrage puis, s'il est positif, les actions.

A ce stade, le prototype ne permet pas de modéliser toutes les actions, en particulier l'apprentissage de règles ou la gestion d'état des connexions.

5 Conclusion

NoD est un outil déjà déployé avec succès dans un cadre industriel, mais il repose sur des optimisations manuelles complexes des programmes. Octant introduit une couche d'abstraction et de nouvelles optimisations formalisées, permettant une utilisation plus facile et sûre par des exploitants.

Références

[1] Sylvain Afchain & Nicolas Planel (2016) : *Skydive, Real-Time Network Topology and Protocol Analyzer.* In : *OpenStack Summit.*

[2] Nuno P. Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman & George Varghese (2015) : *Checking Beliefs in Dynamic Networks.* In : *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, USENIX Association, Oakland, CA, pp. 499–512.

[3] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker & Jonathan Turner (2008) : *OpenFlow : Enabling Innovation in Campus Networks. SIGCOMM Comput. Commun. Rev.* 38(2), p. 69–74.

[4] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J. Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon & Martín Casado (2015) : *The Design and Implementation of Open vSwitch.* In : *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15, Oakland, CA, USA, May 4-6, 2015*, USENIX Association, pp. 117–130.