CNRS - INP - UT3 - UT1 - UT2J
Institut de Recherche en Informatique de Toulouse

L'humain et son environnement au cœur de l'Informatique

ACTES DES 19ÈMES JOURNÉES SUR LES

APPROCHES FORMELLES DANS L'ASSISTANCE AU DÉVELOPPEMENT DE LOGICIELS

AFADL 2020

Édités par David Delahaye et Ileana Ober

IRIT/RR—2020--05--FR
ISBN 978-2-917490-30-3

AVANT PROPOS

Initié il y a plus d'une vingtaine d'années, l'atelier Approches Formelles dans l'Assistance au Développement de Logiciel arrive à sa 19ème édition et se propose de continuer à rassembler des acteurs académiques et industriels intéressés par la mise en œuvre des techniques formelles aux divers stades du développement des logiciels et/ou des systèmes. C'est pour cela que notre atelier a toujours été organisé conjointement avec d'autres ateliers, journées ou conférences, afin de permettre à notre communauté de se rassembler le plus largement possible. Pour sa 19ème édition, l'atelier AFADL était prévu en même temps que les Journées Nationales du GDR GPL (Groupement de Recherche Génie de la Programmation et du Logiciel), qui devaient avoir lieu à Vannes du 19 au 20 juin 2020.

La crise sanitaire que nous avons traversée due à la pandémie de Covid-19, a fait que l'organisation de ces journées a été suspendue et reportée à l'année prochaine. Pour AFADL, lorsque le monde s'est arrêté, les soumissions des papiers avaient déjà été reçues. Avec l'accord du comité de pilotage AFADL et afin d'offrir une diffusion convenable aux travaux en cours, nous avons choisi de poursuivre le processus de relecture. C'est ainsi que ces actes ont pu voir le jour avec en particulier 8 contributions. Les papiers publiés dans ce volume seront présentés lorsque notre conférence pourra effectivement avoir lieu, c'est-à-dire l'année prochaine en juin 2021 à Vannes en même temps que les futures journées du GDR GPL.

La recherche ne s'arrête pas à cause d'une crise sanitaire et nous sommes persuadés que notre communauté sortira de jolis résultats dans l'année qui suit. Pour profiter de ces avancées, un nouvel appel à contributions sera lancé début 2021 et nous conduirons un nouveau processus de relecture, qui donnera lieu à un nouveau volume des actes AFADL. AFADL 2021 sera à plusieurs titres un cru remarquable. En particulier, il offrira une vitrine pour nos travaux durant deux années.

Pour finir, nous profitons de cette occasion pour remercier les membres de notre comité de programme pour leur attentif travail de relecture, pour avoir accepté de travailler dans les circonstances particulières de cette année et par anticipation, pour leur engagement pour la prochaine étape de relecture en 2021.

Bonne lecture à tous et nous espérons vous voir nombreux à Vannes en juin 2021,

David Delahaye
Ileana Ober

PRÉSIDENTS DU COMITÉ DE PROGRAMME

David Delahaye (LIRMM, Université de Montpellier)
Ileana Ober (IRIT, Université Toulouse 3)

COMITÉ DE PROGRAMME

Yamine Aït Ameur (IRIT, ENSEEIHT)
Béatrice Bérard (LIP6, Sorbonne Université)
Sandrine Blazy (University of Rennes 1 - IRISA)
David Deharbe (ClearSy System Engineering)
David Delahaye (LIRMM, Université de Montpellier)
Catherine Dubois (ENSIIE-Samovar)
Marc Frappier (Université de Sherbrooke)
Alain Giorgetti (FEMTO-ST Institute, Univ. of Bourgogne Franche-Comté)
Olivier Hermant (MINES ParisTech)
Mathieu Jaume (Sorbonne Université - UPMC - CNRS LIP6 UMR)
Florent Kirchner (CEA LIST)
Régine Laleau (Paris Est Créteil University)
Thomas Lambolais (LGI2P)
Pascale Le Gall (CentraleSupelec)
Yves Ledru (Laboratoire d'Informatique de Grenoble - Université Grenoble Alpes)
Nicole Levy (Cedric, CNAM)
Delphine Longuet (Univ. Paris-Sud, LRI)
Micaela Mayero (LIPN, Université Paris 13)
Stephan Merz (Inria Nancy)
David Monniaux (CNRS / VERIMAG)
Ileana Ober (IRIT, Université Toulouse 3)
Iulian Ober (IRIT, Université Toulouse 2)
Ioannis Parissis (Univ. Grenoble Alpes - Grenoble INP)
Pascal Poizat (Université Paris Nanterre et LIP6)
Marc Pouzet (LIENS)
Jean-Baptiste Raclet (IRIT)
Vlad Rusu (INRIA)
Nicolas Stouls (CITI/INSA Lyon)
Laurent Voisin (Systerel)
Virginie Wiels (ONERA / DTIM)

Tables de matières

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

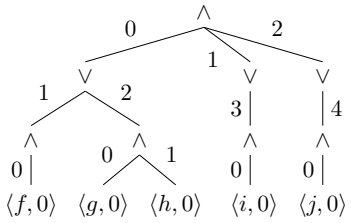
```
P(X0,Y0) :- Q(Y0,X0).
P(X1,Y1) :- f(X1,Y1).
Q(X2,Y2) :- P(X2,Y2).
```

(a) Programme Datalog récursif

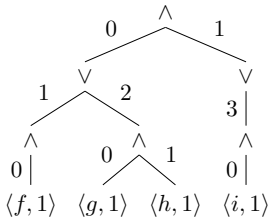
```
P(X0,Y0) :- Q(X0,Y0), R(X0,Y0), S(X0).
Q(X1,Y1) :- f(X1,Z1).
Q(X2,Y2) :- g(X2,Y2,Z2), h(X2,Y2,Z2).
R(X3,Y3) :- i(X3,Y3).
S(X4)    :- j(X4).
```

(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.

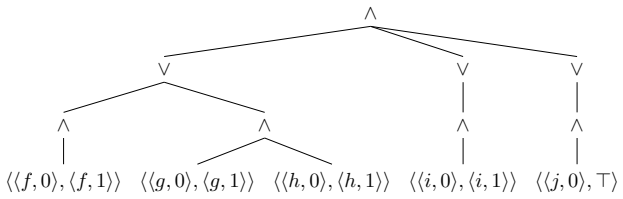


FIGURE 3 – Analyses combinées

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.

Vérification de Recommandations sur les Objets Connectés

Elliott Blot, Sébastien Salva
LIMOS, Université Clermont Auvergne, France
Email: elliott.blot@uca.fr, sebastien.salva@uca.fr

Résumé

Cet article présente SMProVer, une méthode de Model Checking permettant d'aider à l'audit de sécurité des objets connectés, en vérifiant qu'ils suivent un ensemble de recommandations. La méthode utilise un outil permettant de générer le modèle de chaque objet du système, puis nous proposons une technique aidant les auditeurs à instancier des propriétés LTL représentant les recommandations, à partir des modèles générés. Finalement, les propriétés LTL instanciées sont vérifiées sur chaque modèle par un solveur.
Mots clés : Objets Connectés ; Model Checking ; Sécurité

1 Introduction

Les techniques de Model Checking consistent à vérifier des propriétés définies par un expert sur un modèle représentant un système à analyser. Ces techniques ont montré qu'elles étaient efficaces lors d'audit de securité, en verifiant des propriétés de sécurité sur les modèles [1]. Cependant, nous avons constaté que dans la majorité des cas, un expert doit produire à la main le modèle représentant le système, ainsi que les propriétés à vérifier sur celui-ci, ce qui est long et source d'erreurs. De plus, les propriétés écrites par l'expert dépendent grandement du modèle construit précédemment, ce qui signifie qu'il est nécessaire de récrire à nouveau les propriétés si nous souhaitons les vérifier sur d'autres modèles.

Dans cet article nous présentons la méthode SMProVer (Security Measures Properties Verification), une méthode de Model Checking ayant pour but d'aider à l'audit de sécurité des objets connectés, en vérifiant si les objets d'un système suivent des recommandations. L'objectif de cette méthode est de simplifier la production des modèles des objets, et propriétés à vérifier sur chacun d'entre eux. L'écriture des modèles des objets est simplifiée à l'aide d'un outil d'inférence de modèle, qui produit un modèle pour chaque objet du système, aussi appelé composant, à partir d'un log réseau du système. Ensuite, au lieu de produire pour chaque modèle un ensemble de propriétés à partir de l'alphabet du modèle, nous proposons d'enrichir les modèles existants avec l'alphabet de propriétés déjà écrites à l'avance par un expert, puis d'instancier ces propriétés pour chaque modèle à l'aide

d'un algorithme, afin de faire correspondre les variables des propriétés aux données présentes dans le modèle. Cela permet de n'avoir à écrire qu'une seule fois les propriétés, pour les utiliser sur plusieurs systèmes différents.

Notre objectif n'étant pas de vérifier si le modèle contient des vulnérabilités mais de vérifier si les composants du système ont un comportement exprimant que des recommandations sont suivies pour protéger les composants, les propriétés que nous allons vérifier sur nos modèles expriment ces recommandations. L'organisation ENISA propose plusieurs documents proposant des recommandations pour les objets connectés dans divers contextes. Nous avons choisi de nous concentrer sur le contexte d'infrastructures avec informations critiques [2]. Ce document présente 57 recommandations que les composants doivent suivre, notées de GP-TM-1 à GP-TM-57. Nous avons déterminé qu'il était possible de vérifier sur des modèles des communications 11 d'entre elles étant liées au communications du système.

La suite de cet article est organisée de la manière suivante : la Section 2 présente le type de propriétés utilisées par notre méthode. La description de la méthode est donnée en Section 3, et enfin nous concluons en Section 4.

2 Property Types

A partir des recommandations proposées par l'ENISA [2], nous avons écrit 11 propriétés. Ces propriétés ont été obtenues en reformulant en formule LTL les recommandations à l'aide de mots clés, qui sont des propositions atomiques, ou des prédicats composés de variables. L'ensemble des mots clés est noté KW, et l'ensemble des variables utilisées dans les prédicats est noté X. Nous appelons ces formules LTL des *property types*.

Définition 1 (Property type) *Une* property type Φ *est une formule LTL composée de propositions atomiques et/ou de prédicats dans* KW. *L'ensemble des* property types *est noté* $\mathscr{P}$.

Key-word	Description
$from(c)$	Le message est envoyé par c.
$to(c)$	Le message est envoyé à c.
$request$	Le message est une requête.
$output$	Le message est une sortie.
$sensitive(x)$	Le paramètre x contient des données sensible.
$encrypted(x)$	L'affectation du paramètre x est chiffrée.
$loginAttempt(c)$	Tentative d'authentification avec c.
$authenticated(c)$	Authentification réussi avec c.
$validResponse$	Réponse de succès.
$unavailable$	Le composant ayant reçu la requête est indisponible.

TABLE 1 – Exemples de mots clés utilisés dans les *property types*.

La Table 1 donne des exemples de mots clés que nous pouvons trouver dans nos *property types*. Voici quelques exemples de *property types* que nous avons écrit à l'aide de ces mots clés, où $\mathcal{G}$ est l'opérateur LTL *globally* et $\mathcal{U}$ est l'opérateur *until* :

— GP-TM-38 : $\mathcal{G}(sensitive(x) \rightarrow encrypted(x))$
— GP-TM-42 : $\mathcal{G}((\neg(validResponse \wedge to(comp) \wedge \neg loginAttempt(comp))\,\mathcal{U}$
$(authenticated(comp)))\wedge$
$(\neg(request \wedge to(comp) \wedge \neg loginAttempt(comp))\,\mathcal{U}(authenticated(comp))))$
— GP-TM-48 :
$\neg\,\mathcal{G}((from(dep) \wedge unavailable) \rightarrow \neg(\neg output\,\mathcal{U}(output \wedge unavailable)))$

La première propriété exprimée par GP-TM-38, vérifie que toute donnée sensible x dans le réseau est chiffrée. La seconde nommée GP-TM-42 exprime la propriété qui vérifie que tout échange entre composants est précédé par une authentification. Autrement dit, le composant ne doit pas envoyer de requête ou de réponse de succès pour autre chose qu'une tentative d'authentification, tant qu'il n'y aura pas eu d'authentification entre les deux acteurs. La troisième propriété, exprimée par GP-TM-48 vérifie que le composant analysé ne devient pas indisponible après une tentative de communication avec une dépendance ne répondant pas.

3 SMProVer

La méthode SMProVer a pour but de vérifier si des recommandations sont bien suivies par chaque composant du système, à partir d'un ensemble de *property types* représentant ces recommandations, et d'un log contenant les communications entre les composants du système. Pour cela, SMProVer utilise dans un premier temps le log réseau du système afin de générer pour chaque composant un LTS (Labelled Transition System) représentant ses communications. Ces modèles sont ensuite transformés, afin de faire apparaître dans leurs transitions les mots clés présents dans les *property types*. Finalement, pour chaque modèle transformé, les *property types* sont instanciées afin de correspondre aux dépendances et données du composant modélisé. Chacune de ces étapes est détaillée ci-dessous.

3.1 Apprentissage du Modèle

Nous avons proposé dans [3] une méthode appelé CkTail (Communicating system kTail), permettant d'inférer les modèles des composants d'un système communicant, à partir d'un log des communications du système complet. CkTail génère les modèles de chaque composant en analysant les communications entre les composants du système. Plus de détails sur son fonctionnement sont donnés dans [3].

Comme notre processus utilise CkTail pour générer les modèles, les hypothèses sur le système sont les mêmes que celles pour CkTail :

— Le log est collecté de manière synchrone, et les messages contiennent un horodatage permettant, de les ordonner.
— Chaque message contenu dans le log contient des paramètres permettant d'identifier la source et la destination du message. De plus un message est identifié soit comme une requête soit comme une réponse.
— Les composants peuvent fonctionner en parallèle, et communiquer les uns

avec les autres. Cependant, chaque composant ne peut avoir qu'une instance lancée à la fois. Les requêtes sont traitées par le composant dans leur ordre d'arrivée, et chaque requête est associée à une réponse et vice versa. Cette hypothèse, associé à la précédente, permet de détecter les différentes sessions dans le log. À noter que les deux derniers points de cette hypothèse peuvent être remplacés par la présence d'un identifiant de session dans les messages.

CkTail infère pour chaque composant c_i du système, un LTS $\mathcal{L}(c_i)$ représentant le comportement de ses communications, ainsi qu'un graphe de dépendance $Dg(c_i)$ représentant ses dépendances avec les autres composants du système. Par exemple dans la Figure 1, comme $c1$ envoie une requête à $c2$, le graphe de dépendance de $c1$ montre qu'il est dépendant de $c2$. Un LTS est composé d'états, et de transitions étiquetées par un ensemble de labels exprimant les communications du composant.

Définition 2 (LTS) *un LTS (*Labelled Transition System*) est un 4-tuple* $\langle Q, q0, L, \rightarrow \rangle$ *où :*

- *—* Q *est un ensemble fini d'états ;* $q0$ *est l'état initial ;*
- *—* L *est un ensemble fini de labels,*
- *—* $\rightarrow \subseteq Q \times (\mathcal{P}(L) \setminus \{\emptyset\}) \times Q$ *est un ensemble fini de transitions (où* $\mathcal{P}(L)$ *est l'ensemble des sous-ensembles de* L*).*

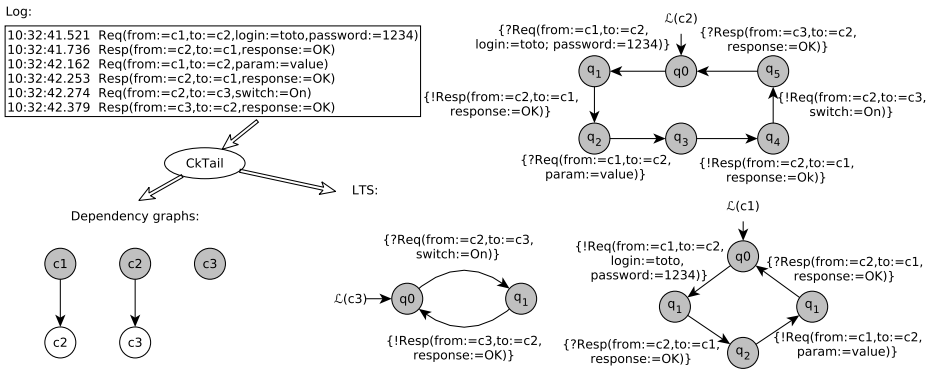


FIGURE 1 – Exemple de résultat obtenu avec CkTail.

La Figure 1 montre un exemple de résultat obtenu en utilisant CkTail. Comme nous pouvons le voir, CkTail produit des LTS n'ayant qu'un label par transition de la forme $a(\alpha)$ avec a un mot et α une affectation de paramètres. Par exemple, le label $Req(from := c2, to := c3, switch := On)$ est composé du mot "Req" suivi d'affectations de paramètres exprimant les composants agissant, et l'affectation de paramètre d'une commande.

Les *property types* ne peuvent pas être utilisées telles quelles sur ces LTS, elles devront être rendues concrètes en instanciant les variables de X des prédicats avec les affectations de paramètres ou composants correspondants pour chaque modèle.

Mais pour connaître ces valeurs, nous allons d'abord devoir transformer nos modèles afin de faire apparaître les propositions atomiques correspondant à ces prédicats dans les labels de nos LTS.

3.2 Transformation du Modèle

SMProVer a pour but de vérifier si un LTS $\mathcal{L}(c_i)$ représentant le comportement du composant c_i satisfait des propriétés LTL. Cela requiert que les formules LTL et les labels du LTS partagent les mêmes propositions atomiques. Généralement, un expert écrit les formules LTL en se basant sur l'alphabet du LTS, ce qui est une tache longue et difficile. Le but de notre approche est de simplifier cette tache, en transformant le modèle du composant $\mathcal{L}(c_i)$ en un nouveau modèle $\mathcal{L}'(c_i)$ contenant les mots clés de l'ensemble KW utilisés pour écrire nos *property types*, dans les labels des transitions. Les propriétés n'ont ainsi qu'à être écrites une seule fois pour pouvoir être vérifiées sur tous les composants ou systèmes.

Pour ajouter ces mots clés il faut analyser et interpréter les transitions de $\mathcal{L}(c_i)$, ce qui nécessite toujours d'avoir une connaissance experte du système. Cependant, nous proposons d'automatiser ce processus à l'aide d'un système expert, utilisant les transitions de $\mathcal{L}(c_i)$ comme base de connaissance, et des règles créées par un expert. Nous pensons qu'il est plus simple de produire ces règles, que de devoir produire des formules LTL en se basant sur l'alphabet des LTS. Les règles du système expert ont la forme suivante : *quand* des conditions sur les transitions sont satisfaites, *alors* des actions sont faites sur ces transitons.

```
rule "authentication"
  when
    t1: Transition(isRequest(), contains("login"), contains("password"))
    t2: Transition(isResponse(), isValid(),
        sourceState(t2)=targetState(t1), from(t2)=to(t1), from(t1)=to(t2))
  then
    t1.addKeyWord("loginAttempt(" + getContact() + ")");
    t2.addKeyWord("authenticated(" + getContact() + ")");
end
```

FIGURE 2 – Exemple de règle de transformation.

Un exemple de règle est donné dans la figure 2. Cette règle nommée *"authentication"* est appliquée lorsque le modèle contient 2 transitions consécutives $t1$ et $t2$, où la première $t1$ est une requête contenant les paramètres "login" et "password", et la seconde $t2$ est une réponse de succès de la requête précédente. Quand cette règle est appliquée, elle ajoute la proposition $loginAttempt(c)$ à la première transition $t1$, et $authenticated(c)$ à la seconde $t2$, avec c le composant avec lequel communique le composant modélisé (obtenu avec la fonction $getContact()$). Cette règle peut être appliquée seulement si les paramètres "login" et "password" apparaissent dans le log lors d'une authentification.

Un exemple de modèle transformé est donné dans la Figure 3. Nous pouvons

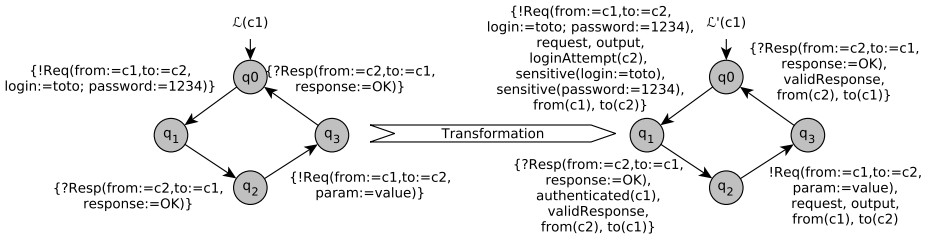


FIGURE 3 – Example de transformation de modèle.

voir qu'après application de notre système expert sur un LTS $\mathcal{L}(c1)$, nous obtenons un nouveau LTS $\mathcal{L}'(c1)$, où certains labels des transitions sont enrichis de propositions atomiques de KW, ou d'instanciations de prédicats de KW.

3.3 Instantiation des Propriétés

Cette étape a pour but d'instancier les *property types* de $\mathscr{P}$ afin d'obtenir un ensemble de propriétés instanciées qui pourront être vérifiées sur $\mathcal{L}'(c_i)$. Nous appelons une propriété instanciée ϕ une *property type* Φ rendue concrète en substituant les variables de X par des valeurs réelles. Les variables de X prennent leurs valeurs dans l'ensemble $C \cup P$, avec C l'ensemble des composants du système et P l'ensemble des affectations de paramètres dans les messages. Ces dernières peuvent être trouvées dans les mots clés des labels des transitions de $\mathcal{L}'(c_i)$. L'ensemble des propriétés instanciées pour le LTS $\mathcal{L}'(c_i)$ est noté $\mathscr{P}(\mathcal{L}'(c_i))$.

Définition 3 (Propriété Instanciée) *Une propriété instanciée ϕ d'une* property type Φ *est une formule LTL obtenue par l'instanciation des prédicats de* Φ.

Les fonctions qui instancient une *property type* en propriété instanciée en associant les variables à des valeurs sont appelées fonctions d'instanciation. A partir d'une *property type* Φ, une fonction d'instanciation rend les variables de Φ concrètes et produit une propriété instanciée ϕ.

Définition 4 (Fonction d'instanciation) *Soit $X' = \{x_1, ..., x_n\} \subseteq X$ un ensemble fini de variables. Une fonction d'instanciation est une fonction $b : X' \rightarrow (C \cup P)^n$, qui assigne une valeur dans $C \cup P$ à chaque variable de X'.*

Cette étape est implémentée par l'algorithme 1 qui prend en entrée un LTS $\mathcal{L}'(c_i)$, l'ensemble de *property types* $\mathscr{P}$ et produit l'ensemble des propriétés instanciées noté $\mathscr{P}(\mathcal{L}'(c_i))$. l'algorithme commence par construire le domaine des variables $comp$ et $deps$ utilisées dans certaines *property types*. $Dom(comp)$ se réfère à tous les composants du système excepté c_i. $Dom(deps)$ contient les composants dépendants de c_i, et est calculé en couvrant le graphe de dépendance de c_i. Pour chaque *property type* Φ, l'algorithme commence par couvrir les transitions $q_1 \xrightarrow{L} q_2$ de $\mathcal{L}'(c_i)$ et labels de L. Si un label $P(v_1, ..., v_k)$ est une instanciation d'un

prédicat $P(x_1, ..., x_k)$ de Φ, alors les valeurs affectées aux variables $x_1, ..., x_k$, sont ajoutées aux ensembles $Dom(x_1), ..., Dom(x_k)$. Si à la fin de ce processus, il reste des variables n'ayant aucune valeur possible, Φ ne peut pas être instanciée et l'utilisateur est alerté. Sinon, l'algorithme rend Φ concrète en appliquant toutes les fonctions d'instanciation possible de $D^{X'}$, avec $D = Dom(x_1) \times ... \times Dom(x_n)$. Pour chaque fonction d'instanciation, une nouvelle propriété instanciée ϕ est générée à partir de Φ.

Algorithme 1 : Instanciation des *property types*

entrée : LTS $\mathcal{L}'(c1)$, ensemble de *property types* $\mathscr{P}$
sortie : ensemble de propriétés instanciées $\mathscr{P}(\mathcal{L}'(c1))$

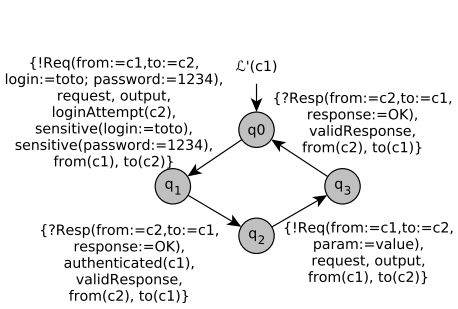
```
1  Dom(comp) = C \ {c1};
2  Calculer Dom(deps) à partir de Dg(c1);
3  X' := ∅;
4  foreach Φ ∈ 𝒫 do
5      foreach l ∈ L with s1 --L--> s2 ∈→_{L'(c1)} do
6          if l := P(v1, ..., vk) et P(x1, ..., xk) est un predicat de Φ then
7              Dom(xi) = Dom(xi) ∪ {vi}(1 ≤ i ≤ k);
8              X' := X' ∪ {x1, ..., xk};
9      if X' est différent de l'ensemble des variables des prédicats de Φ then
10         retourner une alerte;
11     else
12         D := Dom(x1) × ··· × Dom(xk) avec X' = {x1, ..., xn};
13         foreach fonction d'instanciation b ∈ D^{X'} do
14             φ := b(Φ);
15             𝒫(L'(c1)) := 𝒫(L'(c1)) ∪ {φ};
```

Par exemple sur le modèle $\mathcal{L}'(c_i)$ de la Figure 3, avec $Dom(x) = \{login := toto, password := 1234\}$, et en appliquant les fonctions d'instanciation correspondantes, c.a.d $\{x \rightarrow login := toto, x \rightarrow password := 1234\}$ sur la *property type* GP-TM-38 $\mathcal{G}(sensitive(x) \rightarrow encrypted(x))$, nous obtenons les deux propriétés instanciées suivantes :

— GP-TM-38($login := toto$) :
$\mathcal{G}(sensitive(login := toto) \rightarrow encrypted(login := toto))$
— GP-TM-38($password := 1234$) :
$\mathcal{G}(sensitive(password := 1234) \rightarrow encrypted(password := 1234))$.

3.4 Model Checking

Le solveur prend en entrée le modèle transformé $\mathcal{L}'(c_i)$, les propriétés instanciées $\mathscr{P}(\mathcal{L}'(c_i))$, et vérifie chaque propriété sur le modèle. Si toutes les propriétés sont satisfaites, alors le composant suit les recommandations. Dans le cas contraire, cela signifie que le composant c_i ne suit pas les recommandations associées, et que le système peut être vulnérable. Le composant c_i est sûr si et seulement si : $\forall \phi \in \mathscr{P}(\mathcal{L}'(c_i)) : \mathcal{L}'(c_i) \models \phi$. Si une propriété n'est pas satisfaite, un contre exemple est donné par le solveur permettant d'investiguer la vulnérabilité trouvée.




```
Property GP-TM-38(login:=toto) is false:
counterexample:
-q0 -> q1
  !Req(from:=c1,to:=c2,login:=toto; password:=1234),
  request,
  output,
  loginAttempt(c2),
  sensitive(login:=toto),
  sensitive(password:=1234),
  from(c1),
  to(c2)
-q1 -> q2
...

Property GP-TM-42(c2) is true

Property GP-TM-48(c2) is true
```

FIGURE 4 – Exemple de résultats de vérification.

La Figure 4 montre un exemple de résultat obtenu par le solveur NuSMV. Dans cet exemple, la propriété GP-TM-38 instanciée avec la valeur $login := toto$ est invalidée par le solveur, car la transition $q0 \rightarrow q_1$ contient pour donnée sensible $login := toto$, mais cette donnée n'est pas chiffrée. Le composant est donc vulnérable car il communique des données sensibles en clair.

4 Conclusion

Nous avons introduit SMProVer, une méthode pour analyser la sécurité des objets connectés à partir de log réseau du système complet. Ce processus a pour but de vérifier certaines propriétés représentant des recommandations, sur les modèles des composants générés automatiquement. La méthode a été implémentée intégralement afin d'être testée sur 3 logs de système d'objets connectés. Les modèles obtenus comptabilisent plus de 1000 états et transitions, les rendant difficile à analyser manuellement. Notre implémentation a mis moins d'une trentaine de minutes pour analyser chaque système. Nous avons déterminé que 97,4% des recommandations que SMProVer trouve implémentées le sont, et 89,1% des recommandations que SMProVer ne trouve pas implémentées ne le sont effectivement pas. En terme de perspectives, nous prévoyons de faire plus d'expérimentations pour évaluer notre méthode.

Références

[1] R. W. Ritchey and P. Ammann, "Using model checking to analyze network vulnerabilities," in *Proceeding 2000 IEEE Symposium on Security and Privacy. S&P 2000.* IEEE, 2000, pp. 156–165.

[2] ENISA, "Baseline security recommendations for iot in the context of critical information infrastructures," 2017.

[3] S. Salva and E. Blot, "Cktail : Model learning of communicating systems," in *Proceeding of the 15th International Conference on Evaluation of Novel Approaches to Software Engineering*, 2020.

Exploration de Scénarios de Systèmes Cyber-Physiques pour l'Analyse de la Menace

Tithnara Nicolas Sun, Luka Le Roux, Ciprian Teodorov, Philippe Dhaussy

Résumé

La cybersécurité des systèmes est un besoin vital depuis que l'industrie se dirige vers l'automatisation, que ce soit dans les systèmes cyber-physiques ou dans "l'industrie du futur". Il est désormais nécessaire d'envisager la sécurité comme une problématique continue et accompagnant le système tout au long de son cycle de vie. Les méthodes de modélisation actuelles se focalisent généralement sur le système à un instant précis, que ce soit en conception ou en pentest. Cependant, ces analyses ne tiennent pas compte de l'évolution du système et ne garantissent donc pas de résultats pérennes. Dans cet article, nous proposons un environnement de modélisation et de simulation de scénarios d'attaque prenant en compte l'aspect opérationnel du système. Pour cela nous introduisons un langage de scénarios exécutable. Ce langage permet à la fois d'enrichir une modélisation statique du système d'une part et de modéliser le comportement de l'attaquant d'autre part. Nous illustrons notre approche sur une station de pompage. Nous montrons comment un expert peut capturer d'une manière abstraite sa compréhension du système afin de la confronter à une modélisation des capacités de l'attaquant. Ceci permet d'exhiber des analyses formelles de sécurité sur le système sous attaque.

1 Introduction

Les systèmes industriels se complexifient de plus en plus, du fait de l'automatisation croissante des procédés. Cela implique une dépendance grandissante en ces processus automatiques. C'est pourquoi il est crucial de garantir la sécurité de fonctionnement des systèmes face à la menace cyber [3]. L'ingénierie dirigée par les modèles préconise un ensemble de techniques permettant de répondre à ce besoin. Les systèmes sont représentés par des modèles sur lesquels peuvent s'appliquer des analyses de vérification et de validation formelles [8].

Toutefois, les méthodes de modélisation actuelles connaissent certaines limitations en sécurité. Premièrement dans un contexte *ad hoc*, c'est-à-dire sur un système existant , du point d'un attaquant opportuniste, la connaissance du système est partielle. Du point de vue de l'attaquant, le modèle du système ne peut pas être détaillé à un niveau d'abstraction trop bas. L'utilisation de modèles de spécification du système est donc à remettre en question. D'autant plus que le système possède un comportement dynamique lié à son fonctionnement. Cet aspect dynamique implique des changements de phases qui peuvent induire des faiblesses exploitables

par un attaquant pour s'introduire dans le système et/ou causer des dommages. Deuxièmement, les méthodes d'analyse ne sont généralement pas pérennes. Les résultats d'analyses ne sont valides que sur la configuration courante du système. Le système, au cours de son cycle de vie, peut changer de configuration lors de changements volontaires (maintenance) ou involontaires (panne). Ces variations mettent en doute la validité des analyses sur la durée.

Afin de résoudre ces problèmes, nous proposons un environnement de modélisation et de simulation de scénarios d'attaque. Pour ce faire, nous nous basons sur trois processus distincts sur un environnement de modélisation et de simulation : la modélisation de l'architecture du système, la modélisation du comportement nominal du système et la modélisation de l'attaquant. Cette approche modulaire permet de spécialiser séparément chaque aspect du problème et donc de concevoir des outils adéquats. Notre approche est mise en oeuvre avec le langage de modélisation statique Pimca pour l'architecture du système [10] associé à un langage de scénarios exécutable dédié Target System Modeling (TSM). Nous présenterons les langages ainsi que leur utilisation avec le model-checker OBP2 (Observer-Based Prover 2) pour les analyses formelles de sécurité. Le langage de scénarios exécutable que nous introduisons permet de capturer le comportement nominal du système et le comportement de l'attaquant. À partir de cet environnement, nous montrons comment un expert peut représenter sa compréhension du système de manière abstraite et comment produire des analyses de sécurité.

Dans cet article, nous introduirons les langages utilisés en détaillant le langage de comportement. En parallèle, nous illustrons notre approche sur le cas d'étude d'une station de pompage. Puis nous présentons nos résultats d'analyse. Enfin nous concluons en mettant en perspective notre approche avec la littérature.

2 Capture du fonctionnement du système

Dans le cadre de l'analyse de la menace de systèmes industriels, nous modélisons la structure du système étudié avec le langage Pimca [10], puis nous modélisons le comportement nominal du système à l'aide de notre langage de scénarios TSM. Enfin, nous modélisons également le comportement de l'attaquant avec le langage TSM. Notre approche est outillée avec l'environnement Openflexo, le cadre logiciel est disponible en ligne. [1] La section 2.1 introduit le cas d'étude qui illustrera l'approche. Nous présentons Pimca en 2.2 et TSM en 2.3.

2.1 Station de pompage

La station de pompage, fig.1, est un système cyber-physique contrôlant automatiquement le niveau d'un réservoir d'eau grâce à un PLC (Programmable Logic Controller). Une vanne électrique d'entrée alimente le réservoir en eau, tandis qu'une vanne manuelle de sortie, connectée à une pompe électrique, vide le réservoir. La vanne manuelle est actionnable par un technicien. La vanne électrique et la pompe sont directement contrôlées par le PLC qui relève le niveau d'eau du

1. https ://research.openflexo.org/CTA.html

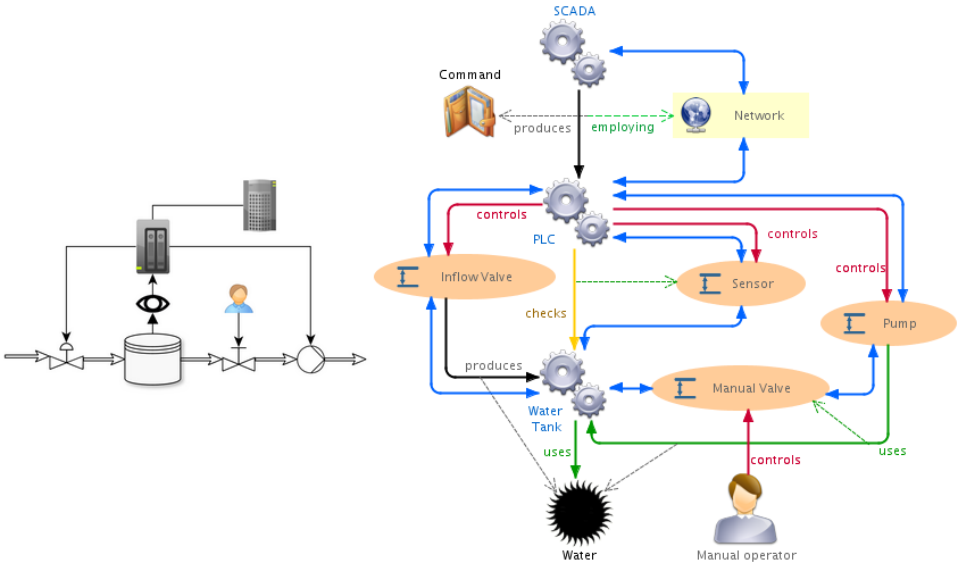


FIGURE 1 – Station de pompage : structure (gauche), modèle Pimca (droite)

réservoir à travers un capteur. De plus le PLC relaie régulièrement les mesures du niveau d'eau à une centrale SCADA (Supervisory Control And Data Acquisition) sur le réseau du site industriel. Le comportement du PLC suit les commandes suivantes : (i) ouvrir la vanne d'entrée et éteindre la pompe si le niveau d'eau atteint un seuil bas, (ii) fermer la vanne d'entrée et allumer la pompe si le niveau d'eau atteint un seuil haut, (iii) chaque mesure de niveau est envoyée au SCADA.

2.2 Pimca

Afin de conduire des analyses de sécurité, une modélisation adaptée du système est nécessaire. Pimca [10] est un langage graphique de modélisation système pour la sécurité qui met en lumière la surface d'attaque [5], l'ensemble des points d'interactions avec le système pour l'attaquant. C'est un langage à haut niveau d'abstraction qui satisfait le contexte de modélisation *ad hoc*. Pimca modélise le système en différents composants pourvus de comportements (*machinerie*). Les éléments du système susceptibles d'être cibles d'un attaquant sont représentés par des *ressources*. Ces composants sont liés entre eux via des *relations* expressives.

À partir de la description du système, nous identifions les différents composants qui se traduisent en éléments de modèle Pimca, fig.1. Le réservoir, les vannes, la pompe, le capteur, le technicien, le PLC et le SCADA présents dans le schéma, fig.1, sont capturés par des machineries car ils possèdent un comportement. De plus, nous ajoutons au modèle le réseau du site industriel à travers lequel communiquent le SCADA et le PLC, car c'est un point d'échange actif du système et donc une machinerie. Enfin, nous modélisons deux cibles potentielles pour un attaquant, à savoir les instructions suivies par le PLC et l'eau qui circule dans le système.

Ensuite nous identifions les différentes relations du modèle Pimca. Les connexions entre les composants représentés dans le schéma initial sont modélisées

par la relation bidirectionnelle *échange*. Pour les relations de plus haut-niveau, nous pouvons distinguer différentes classes : (i) le *contrôle* que le PLC exerce sur la vanne d'entrée, la pompe et le capteur, (ii) le *contrôle* exercé par le technicien sur la vanne manuelle, (iii) la *production* d'eau au regard du système de la vanne d'entrée pour le réservoir, (iv) l'*utilisation* de l'eau du réservoir par la pompe, (v) la *production* de commandes du SCADA au PLC, (vi) l'*utilisation* de l'eau par le réservoir et (vii) la *vérification* du niveau du réservoir par le PLC.

Ce modèle statique capture la structure du système et permet des analyses de sécurité telles que la déduction de surface d'attaque présentée dans [10]. En exploitant les relations expressives, il permet un raisonnement itératif sur les cibles intermédiaires et les points d'entrées potentiels . Toutefois pour conduire des analyses de sécurité tenant compte des aspects dynamiques du système, il est nécessaire de capturer le comportement dynamique.

2.3 Target System Modeling

Afin de nous focaliser sur cette problématique, nous introduisons un nouveau langage exécutable dédié, Target System Modeling (TSM). À travers TSM, nous spécifions ces comportements pour exécuter, simuler et produire des analyses. ***Comportement nominal :*** Dans Pimca, les *machineries* sont définies comme les éléments pourvus de comportement. Le comportement du système est modélisé par un ensemble de machines à états synchronisées. L'ensemble des comportements d'un composant constitue une machine à états qu'on appelle une unité d'exécution. Les unités d'exécutions sont reliées entre elles (une-à-une), à la manière des machineries de Pimca, à travers des canaux de synchronisation. Ensuite, nous capturons la configuration du système global dans un ensemble de variables dont certaines sont globales et d'autres sont internes à une unité d'exécution donnée.

TSM utilise un formalisme d'actions similaire au formalisme introduit par Dijkstra [2] pour capturer le comportement au sein d'une unité d'exécution. Une action TSM est définie comme un couple de garde/commande. La garde est une expression booléenne qui détermine si l'action est déclenchable. La commande décrit les effets de l'action. Ceci permet de modéliser séparément chaque composant du système. De plus, pour permettre la communication synchrone entre les différentes unités d'exécutions, une action peut être assignée à un canal de synchronisation. Cela signifie que l'action doit être déclenchée en synchronisation avec une action de l'autre côté du canal en un seul pas atomique. Cependant, ce formalisme ne permet pas la synchronisation de plus de deux unités d'exécution à la fois, c'est pourquoi nous introduisons la notion d'action urgente. Les actions urgentes doivent être déclenchées en priorité par rapport aux autres actions. Ceci permet de gérer les synchronisations entre de nombreuses unités d'exécution en les considérant comme des synchronisations une-à-une urgentes qui se résolvent toutes avant que le système ne puisse évoluer, via des actions non urgentes. Le prototype d'une action est donc : $ID : urgent?\ chanID(?\ |!)([garde])?/(commande;)*$.

Dans le cas de la station de pompage, certaines instances de machineries

(réseau, capteur, vanne manuelle) sont simplement des relais, leur comportement consiste à relayer un message d'un élément émetteur du système à un autre élément récepteur. Par exemple, le niveau d'eau du réservoir est transmis au PLC à travers le capteur. Ceci est capturé par une action synchronisée avec l'émetteur pour réceptionner le message et par une action urgente synchronisée avec le récepteur pour envoyer le message. Concrètement le capteur est une unité d'exécution possédant 2 variables (*waterLevel, isTriggered*) et contenant 2 actions :
-$reception : mesure? / waterLevel := value; isTriggered := true;$
-$emission : urgent\ updatePLC! [isTriggered] / isTriggered := false;$

Le réservoir d'eau possède un niveau qui fluctue en fonction de l'arrivée d'eau depuis la vanne électrique et la sortie depuis la vanne manuelle. De plus, le niveau d'eau est relevé par le capteur lorsqu'il évolue. Ce comportement est capturé par des actions synchronisées pour augmenter le niveau lorsque la vanne électrique envoie de l'eau et diminuer le niveau lorsque la vanne manuelle pompe de l'eau. Le niveau d'eau est relayé au capteur à chaque changement par une action urgente synchronisée avec le capteur. La pompe et la vanne électrique ont une action qui leur permet de déclencher le changement de niveau d'eau dans le réservoir quand ils sont en marche. Leur état de marche est contrôlé par le PLC à travers une action synchronisée. La vanne manuelle peut être fermée ou ouverte par un technicien.

Le PLC a un comportement plus complexe. Sa commande stipule que le niveau d'eau du réservoir doit être maintenu entre un seuil bas et un seuil haut donnés. Pour ce faire, à chaque relevé de niveau d'eau dans l'action synchronisée avec le capteur, le PLC détermine si le système est à un niveau convenable ou s'il approche les valeurs à risques. Ensuite une action urgente synchronisée avec le SCADA, à travers le réseau, permet de relayer cette information. Enfin, si le PLC a déterminé que le niveau atteignait prochainement le seuil haut, deux actions urgentes synchronisées ordonnent à la pompe et à la vanne électrique de se mettre en marche et de s'éteindre respectivement. Dans le cas du seuil bas, le contraire est modélisé. Le SCADA réceptionne les messages du PLC. Il possède une action déclenchée lorsque le niveau est anormal qui met le SCADA en alerte.

À partir de ce modèle de système, nous simulons pas-à-pas le comportement nominal pour vérifier et valider le comportement du système. Pour conduire des analyses de sécurité, il reste toutefois à modéliser le comportement de l'attaquant

Comportement de l'attaquant : En utilisant le langage TSM, nous étendons le modèle pour permettre l'intervention d'un attaquant. Pour cela, nous spécifions des possibilités d'interactions avec les composants du systèmes, c'est-à-dire les unités d'exécutions. Nous définissons ces interactions comme des altérations du comportement nominal. Elles sont modélisées par l'ajout d'actions d'attaque au sein des unités d'exécution. L'objectif de l'attaquant est de causer un dysfonctionnement du système. Nous recherchons les scénarios possibles qui peuvent mener à cet objectif à travers des actions d'attaque. Notre modélisation d'attaquant revient donc à capturer des capacités d'interactions que l'attaquant a avec le système.

Dans le cas de la station de pompage, l'attaquant a deux objectifs : (i) causer un débordement du réservoir et (ii) ne pas déclencher l'alerte du SCADA en cas de

Forcer vanne d'entrée		•					•	•	•		•	•
Fermer vanne manuelle			•						•	•		•
Bloquer pompe				•				•		•	•	
Brouiller réseau					•		•			•	•	•
Couper capteur						•						
Objectif 1	X	X	X	X	X	**O**	X	O	O	X	**O**	**O**
Objectif 2	-	-	-	-	-	**O**	-	X	X	-	**O**	**O**

TABLE 1 – Model-checking de la station de pompage (O : succès, X : échec)

succès du débordement. Voici la liste des actions d'attaque que nous modélisons : (i) forcer l'ouverture de la vanne électrique, (ii) fermer la vanne manuelle, (iii) bloquer la pompe, (iv) brouiller le réseau, (v) couper le capteur.

3 Analyse de sécurité

Le modèle exécutable nous permet de rechercher des scénarios d'attaque en fonction des capacités de l'attaquant grâce au model-checking. En effet, les objectifs de l'attaquant peuvent être formulés en propriétés LTL qui peuvent être vérifiées. Si ces propriétés sont violés alors l'attaquant peut atteindre ses objectifs. Dans notre cas, les propriétés sont les suivantes :

1. "`[] !|waterOverflow|`"
2. "`[] (|waterOverflow| -> <>|scadaAlert|)`"

Les propositions `|waterOverflow|` et `|scadaAlert|` rédigées en TSM expriment respectivement que le réservoir déborde et le SCADA est en alerte.

Notre implémentation exécutable de TSM utilise Java et peut être connectée au model-checker OBP2 pour la simulation et l'analyse du comportement. OBP2 manipule la configuration d'une instance du modèle TSM. Il permet d'évaluer des prédicats et de calculer l'ensemble des transitions tirables vers de nouvelles configurations à partir de la configuration courante. L'interface d'OBP2 permet de simuler pas à pas le comportement du système. OBP2 garde en mémoire la configuration courante ainsi que la trace d'exécution ce qui permet de recharger une configuration antérieure. L'interface permet également de déclencher les transitions tirables depuis la configuration courante pour une exploration manuelle. Par ailleurs OBP2 permet de vérifier des propriétés LTL avec GPSL [1]. Lors de l'exploration automatique impliquant la composition d'automates de Büchi, le model checker explore l'espace d'états du système et indique si une propriété est valide ou non.

Dans un premier temps, nous avons simulé le comportement nominal du système, c'est-à-dire sans qu'aucune action d'attaque ne soit possible pour vérifier que le système fonctionnait sans action malveillante. Ensuite nous avons vérifié les propriétés 1 et 2 sur l'ensemble des combinaisons de capacités d'attaquant possibles pour déterminer les capacités qui lui permettent d'atteindre ses objectifs.

La table 1 montre les résultats de notre analyse avec l'outil OBP2 et l'exploration des configurations du système. Chaque colonne représente une combinaison de capacités de l'attaquant ainsi que le fait que l'attaquant puisse atteindre son

objectif 1 (faire déborder le réservoir) et son objectif 2 (ne pas se faire répérer par le SCADA en cas de débordement). On peut notamment voir que l'attaquant peut atteindre ses deux objectifs avec une seule capacité : "couper le capteur". On peut également noter que les capacités "fermer la vanne manuelle" et "bloquer la pompe" sont équivalentes au regard des objectifs, il est donc inutile pour l'attaquant d'utiliser ces deux capacités. Enfin on peut voir que si l'attaquant ne peut pas couper le capteur, il doit alors brouiller le réseau, forcer la vanne d'entrée et arrêter le flux de sortie d'une manière ou d'une autre pour arriver à ses fins.

Cette analyse du modèle TSM de la station de pompage exhibe un point critique dans le fonctionnement du système. Le capteur est un élément essentiel puisque sa mise hors service permet à l'attaquant d'atteindre ses deux objectifs. Le modèle et les analyses sont disponibles en ligne. [2]

4 État de l'art

L'exploration de scénarios de systèmes cyber-physiques pour l'analyse de la menace est un problème qui alimente de nombreux travaux de recherche.

Certaines approches produisent des analyses d'impacts [7, 9]. Celles-ci considèrent la dynamicité du modèle et permettent de capturer le comportement du système pendant une attaque. Néanmoins, la modélisation de l'attaquant est insuffisante dans le premier cas [7] pour capturer des scénarios d'attaques complexes. Dans le second cas [9], le formalisme d'automates permet de représenter des impacts d'attaques plus complexes au prix d'une modélisation intrusive aux composants (ajouts d'états et transitions) ce qui demande une compréhension poussée du fonctionnement du système et pose des problèmes de pérennité. Il est donc difficile d'appliquer ces approches dans un contexte *ad hoc*.

D'autres approches sont basées sur le formalisme d'arbres d'attaque [4, 8]. Les arbres d'attaques permettent de modéliser des scénarios d'attaque dans un formalisme expressif. Contrairement à notre approche, ces outils mettent en jeu de multiples niveaux d'abstraction, ce qui requiert une compréhension poussée du fonctionnement système et de ses vulnérabilités. Ceci ne convient pas à un contexte de modélisation *ad hoc* et limite l'extensibilité de la modélisation, donc sa pérennité.

5 Conclusion

L'ingénierie dirigée par les modèles peut apporter des réponses pertinentes à la sécurité des systèmes cyber-physiques. Nous proposons un environnement de modélisation et de simulation de systèmes utilisant deux langages : Pimca pour la modélisation structurelle et TSM pour la modélisation comportementale. Notre approche exécutable est implémentée avec OpenFlexo, connectée au model-checker OBP2 et validée sur le cas d'étude de la station de pompage.

L'approche repose sur la complémentarité Pimca-TSM pour répondre au besoin de modélisation *ad hoc* des systèmes. D'une part, elle capture d'une manière abstraite la dynamique du système pour modéliser ses changements de phases.

2. https ://github.com/Lawyne/pimca-tsm-afadl20

D'autre part, elle permet de capturer d'une manière peu intrusive le comportement de l'attaquant. En effet, les actions sont aisément extensibles et se prêtent donc mieux à l'extension de modèles que les formalismes d'automates ou d'arbres.

Il reste néanmoins à étudier la possibilité d'étendre le formalisme TSM avec une couche réflexive[6] nous permettant la modélisation non-intrusive et réutilisable de l'attaquant sur des systèmes plus complexes.

Remerciements : Nous tenons à remercier la Direction Générale de l'Armement (DGA) qui finance en partie ce projet.

Références

[1] M. Brumbulli, E. Gaudin, and C. Teodorov. Automatic Verification of BPMN Models. In *10th European Congress on Embedded Real Time Software and Systems (ERTS 2020)*, Toulouse, France, 2020.

[2] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8) :453–457, 1975.

[3] A. Ginter. The Top 20 Cyberattacks on Industrial Control Systems. Technical report, Waterfall Security Solutions, 2017.

[4] B. Kordy, L. Piètre-Cambacédès, and P. Schweitzer. DAG-based attack and defense modeling : Don't miss the forest for the attack trees. *Computer science review*, 13 :1–38, 2014.

[5] P. K. Manadhata and J. M. Wing. An attack surface metric. *IEEE Transactions on Software Engineering*, 37(3) :371–386, 2011.

[6] N. Papoulias, M. Denker, S. Ducasse, and L. Fabresse. End-user abstractions for meta-control : Reifying the reflectogram. *Science of Computer Programming*, 140 :2 – 16, 2017. Object-Oriented Programming and Systems (OOPS 2015).

[7] E. Peterson. Dagger : Modeling and visualization for mission impact situation awareness. In *MILCOM 2016-2016 IEEE Military Communications Conference*, pages 25–30. IEEE, 2016.

[8] S. Pinchinat, M. Acher, and D. Vojtisek. ATSyRa : an integrated environment for synthesizing attack trees. In *International Workshop on Graphical Models for Security*, pages 97–101. Springer, 2015.

[9] B. Sultan, F. Dagnat, and C. Fontaine. A methodology to assess vulnerabilities and countermeasures impact on the missions of a naval system. In *Computer Security*, pages 63–76. Springer, 2017.

[10] T. N. Sun, B. Drouot, J. Champeau, F. R. Golra, S. Guérin, L. Le Roux, R. Mazo, C. Teodorov, L. Van Aertryck, and B. L'Hostis. A Domain-Specific Modeling Framework for Attack Surface Modeling. In *Proceedings of the 5th International Conference on Information Systems Security and Privacy*, pages 341–348. SCITEPRESS, 2020.

Vérification formelle d'un réseau sur puce : Application de DEv-Promela

A. Khemiri, A. Yacoub et M. Hamri
Aix-Marseille Univ, Université de Toulon, CNRS
LIS, Marseille, France
{abdelhak.khemiri, aznam.yacoub, amine.hamri}@lis-lab.fr

Résumé

Dans ce papier, nous présentons une approche pour vérifier et valider des systèmes dynamiques complexes qui reposent à la fois sur une vérification formelle des propriétés temporelles exprimées et une simulation (tester un scénario, quantifier des variables ou aider à la prise de décision) de la dynamique du système modélisé.

En effet, nous montrons qu'il est possible de combiner un vérificateur et une simulation pour tirer les avantages et combler mutuellement les inconvénients de chacun. Nous nous appuyons sur le langage DEv-PROMELA qui, lui, repose sur deux formalismes développés par deux communautés différentes : PROMELA pour vérifier formellement des systèmes asynchrones et DEVS pour modéliser et simuler des systèmes dynamiques.

Enfin, une étude de cas décrivant un réseau sur une puce électronique est présentée et les résultats obtenus sont analysés et commentés.

1 Introduction

La vérification formelle de systèmes logiciels et de programmes informatiques est un champ de recherche très actif avec des avancées théoriques récentes. Des progrès considérables en termes de méthodes et d'outils ont été réalisés rapprochant de plus en plus les techniques de vérification formelle des développeurs informatiques.

Prônant le rapprochement entre la simulation et la vérification formelle pour vérifier et valider des spécifications de système [6, 5], nous avons proposé le langage DEv-PROMELA (*Discrete Event Process MetaLanguage*) [8]. Ce langage, préservant les propriétés structurelles de PROMELA d'une part et les propriétés temporelles des DEVS d'autre part, décharge l'utilisateur de la construction d'abstractions formelles à partir des spécifications pour démontrer une propriété quelconque. En effet le modèle PROMELA est extrait automatiquement à partir d'une spécification DEv-PROMELA évitant l'introduction de nouvelles erreurs. En revanche, le modèle DEVS extrait automatiquement aussi permet de jouer des scénarios et réaliser des mesures par simulation.

Afin de montrer l'utilité de ce langage pivot et de son cadre de travail, nous décrivons un réseau sur puce électronique et nous étudions ses propriétés par une vérification formelle hybride. Cette étude repose à la fois sur la simulation DEVS et le model checking pour traiter deux difficultés rencontrées lors de la recherche d'erreurs dans des systèmes complexes. Le premier objectif est de repousser le plus loin possible le problème d'explosion combinatoire de l'espace d'états. Enfin, le second objectif vise à réduire la différence (distance) entre le modèle formel de vérification et le code exécutable final.

2 Rappels sur DEv-Promela

DEv-PROMELA (Discrete Event Process MEta LAnguage) est une extension de PROMELA, un langage développé pour la description et la vérification formelle de processus asynchrones. DEv-PROMELA quant à lui est développé pour la description et la vérification de systèmes à événements discrets complexes, en utilisant une technique de vérification formelle à base de model checking et une simulation à événements discrets. Son vérificateur repose sur l'interpréteur SPIN © pour vérifier toute propriété exprimée en LTL (Linear Temporal Logic) et le simulateur DEVS pour générer des traces d'exécution à partir de scénarios exprimés initialement.

Le langage DEv-PROMELA intègre des concepts de la modélisation DEVS permettant de raffiner une spécification PROMELA. En effet, les instructions peuvent être datées, déclenchées suite à l'occurrence d'un événement ou émettre un événement entre des processus. L'intérêt d'un tel langage est qu'il permet d'avoir un modèle unique où une vérification formelle des propriétés du système et une simulation de scénarios par une génération de trace d'exécution peuvent être effectuées. Pour ce faire, deux modèles sont extraits à partir de la description DEv-PROMELA : (1) le modèle PROMELA garantissant la structure du modèle DEv-PROMELA et permettant d'effectuer une vérification formelle, et (2) le modèle DEVS garantissant la dynamique du modèle DEv-PROMELA pour une simulation.

Grâce aux relations de morphisme qui existent entre DEv-PROMELA et PROMELA d'une part, et DEv-PROMELA et DEVS d'autre part, les deux modèles extraits de vérification et de simulation re-décrivent fidèlement le modèle initial DEv-PROMELA. Les conclusions déduites restent vraies pour le modèle DEv-PROMELA à vérifier. Ainsi, toute propriété prouvée au niveau du modèle PROMELA extrait, est une propriété prouvée pour le modèle DEv-PROMELA. De même, toute trace d'exécution générée par le modèle DEVS est générée par le modèle DEv-PROMELA.

Ces règles de construction des modèles PROMELA et DEVS, ainsi que la préservation des propriétés structurelles et dynamiques de chaque modèle sont détaillées dans les travaux de thèse d'Aznam Yacoub [4].

2.1 PROMELA

PROMELA est un langage de spécification de processus informatiques. Il permet la vérification de protocoles synchrones et asynchrones entre processus. Basé sur le langage de garde de Dijikstra [2], sa syntaxe est très proche d'un langage impératif (le langage C, etc.) rendant son utilisation facile par rapport à d'autres langages formels. En effet, ce langage a été développé dans le but d'une transformation possible et facile d'une implémentation vers un modèle de vérification et de réduction d'erreurs de codage.

Bien que PROMELA possède une syntaxe et une sémantique très proche du langage C, la différence réside au niveau de l'abstraction, extraite à partir du code, qui intègre des comportements (exécutions) non-déterministes à l'inverse d'un programme C. Notons qu'une spécification PROMELA se décline en deux parties : (1) une spécification fonctionnelle et comportementale décrivant le système modélisé, et (2) des propriétés à vérifier.

Afin de combler le manque de concepts et d'éléments temporels essentiels à la quantification (mesure) d'une exécution, quelques extensions de PROMELA ont été proposées. Nous citons real-time PROMELA, discrete-time PROMELA et timed PROMELA (une étude comparative de ces extensions est fournie dans [7]). Ces extensions s'accordent sur une représentation discrète du temps qui peut ralentir considérablement une simulation et provoquer des erreurs de précision des résultats obtenus (un événement qui ne se produit pas à la bonne date d'occurrence, etc.). Par la suite, nous préconisons une modélisation DEVS à base d'événements offrant une représentation continue du temps.

2.2 DEVS

DEVS offre un cadre de modélisation et simulation de systèmes dynamiques dirigé par des événements [9]. Il sépare les besoins de modélisation à la charge de l'utilisateur (modélisateur) de la mécanique de simulation réutilisable comme telle dont le but est de reproduire la dynamique du système modélisé. En revanche, la modélisation consiste à décrire un système par un ensemble de composants interconnectés entre eux pour l'envoi et la réception d'événements. Les composants sont décrits à l'aide de modèles atomiques ou couplés. Un modèle atomique est une structure algébrique $M = (X, Y, S, \delta_{ext}, \delta_{int}, \lambda, D)$ qui décrit un comportement par un ensemble d'états (segments) S. En effet, l'état du modèle M évolue dans l'un des deux cas : (1) suite à l'occurrence d'un événement externe $x \in X$ provoquant l'exécution de la fonction $\delta_{ext}()$, ou (2) à la différence entre les dates d'occurrence de deux événements successifs dépassant la durée de vie de l'état courant $D(s)$. Un modèle couplé $MC = (X, Y, D, M_{d \in D}, EIC, EOC, IC, select)$ est un réseau de modèles interconnectés par trois différents types de relations de couplage (EIC, EOC, IC). Un tel modèle est fermé sous couplage, ie., il existe son équivalent en atomique produisant les mêmes comportements. Ainsi, un modèle couplé est légitime à la simulation.

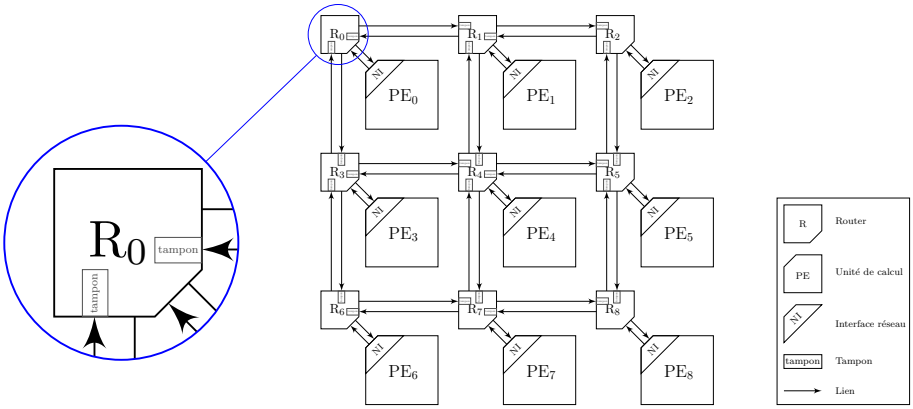


FIGURE 1 – Exemple d'un réseau sur puce maillé en 2 dimensions de taille 3×3

De nombreuses extensions DEVS ont été développées dans le but de répondre à des problèmes particuliers. Parmi elles, nous citons PDEVS pour décire des processus parallèles, DS-DEVS pour décrire des systèmes à structure variable, RT-DEVS pour décrire des systèmes temps réel, etc. Notons aussi que peu de travaux d'analyse formelle des modèles de simulation DEVS ont vu le jour. Bien que ce formalisme ait pour vocation la description de modèles simulables, quelques travaux ont étudié les aspects théoriques de ce formalisme : rapprochement entre DEVS et Timed Automata, analyse statique des modèles DEVS, proposition d'une hiérarchie de formalismes DEVS, etc.

3 Etude de cas : réseau sur puces électroniques

3.1 Description

Un NoC (*Network on Chips*) est un sous-système de communication sur un circuit intégré (puce), entre des coeurs de propriété intellectuelle. Ces différents composants communiquent alors entre eux en échangeant des paquets au travers d'un réseau d'interconnexions, fournissant une structure de communication réutilisable. Figure 1 décrit un exemple de NoC maillé en 2 dimensions (*mesh*) de taille 3×3 avec l'ensemble des éléments qui le compose.

Rappelons que les principaux composants d'un NoC sont : (i) les unités de calcul, (ii) les routeurs qui dirigent les données dans le réseau en accord avec un protocole et une stratégie de routage prédéfinis (iii) les liens qui connectent physiquement les routeurs deux à deux. Un lien peut être unidirectionnel ou bidirectionnel et peut fournir plusieurs canaux virtuels de communication, (iv) les interfaces réseaux dont le but est de séparer et faire l'interface entre le protocole de communication au sein du réseau, et le protocole dans les unités de calcul, et (v) les tampons mémoires qui peuvent être ajoutés en entrée et/ou en sortie des routeurs selon la stratégie de routage mise en place.

L'organisation de l'interconnexion entre les composants au sein d'un réseau est donnée par la topologie de ce dernier. Bien qu'il existe plusieurs topologies, celle de type maillé en 2 dimensions est la plus courante (comme illustré dans Figure 1).

Un des éléments importants du protocole est la manière dont les messages transitent. Dans ce qui suit, nous considérons uniquement la technique de commutation de paquets. Dans cette technique, les unités de calcul s'échangent des messages préalablement divisés en paquets, au travers du réseau. Un paquet correspond alors à la plus petite unité de communication qui contient aussi des informations de routage et de séquençage. Chaque paquet est lui même formé d'un ensemble de *flit* (*FLow control unIT*) qui correspond à la plus petite unité de contrôle de flux sur un lien. Le paquet débute alors par une entête (*header*) qui contient des informations relatives au routage. Le corps du paquet quant à lui contient la charge utile qui correspond au message à transmettre. Enfin, le paquet se termine par une queue, indiquant sa fin.

A cette stratégie de commutation, vient s'ajouter une politique de mémorisation au sein des routeurs. Bien qu'il existe plusieurs politiques de mémorisation, une des plus connues est la mémorisation *wormhole* [1]. Dans cette dernière, les routeurs ne mémorisent pas le paquet en entier avant de le transmettre au routeur suivant. Un *flit* d'entête alors détermine la direction de routage et se transmet en premier lieu. Les autres *flits* le suivent en pipeline.

Un autre élément essentiel du protocole de communication est la stratégie de routage. Cette dernière a pour but de déterminer le chemin à suivre entre la source et la destination d'un paquet. L'un des algorithmes de routage les plus utilisés dans le cas d'une topologie maillé en 2 dimensions est le routage ordonné par dimension X-Y. Il impose que les routeurs acheminent un paquet sur la dimension X puis sur la dimension Y. Cette stratégie a la particularité d'être déterministe (ie., pour un couple (source, destination), le chemin est toujours le même). Plus de détails concernant les réseaux sur puces sont présentés dans [1].

Notons que cette technique de routage est reconnue pour exclure le risque d'interblocage. En mettant les ressources du réseau dans un ordre particulier et en exigeant que les paquets demandent et utilisent ces ressources dans un ordre strictement monotone, l'attente circulaire (condition nécessaire à l'apparition d'un interblocage) est évitée. De nombreuses propriétés peuvent être vérifiées à l'aide de modèles formels, allant de l'évaluation du rendement à la correction de la communication, y compris la vérification fonctionnelle et l'analyse comportementale. Ici, nous examinerons les propriétés de correction de NoCs telles que l'absence d'interblocage.

3.2 Vérification formelle des propriétés temporelles

Afin de s'assurer qu'un modèle M vérifie une propriété p ($M \models p$), il faudra que le modèle en question s'appuie sur une notation mathématique sinon il est impossible démontrer une quelconque propriété.

Nous nous intéressons à la propriété d'absence d'interblocage entre unités de

calcul. Il faudra s'assurer alors qu'un message puisse arriver toujours à sa destination. Le modèle PROMELA décrivant un NoC de taille 3×3 montre qu'il est impossible de vérifier une telle propriété dû à la description très détaillée nécessaire à la simulation engendrant une explosion de l'espace d'états. En effet, les différents algorithmes de parcours du graphe d'exécution, généré à partir de ce modèle, échouent en n'arrivant pas à parcourir l'espace d'états d'une manière exhaustive. Nous avons alors proposé une approche qui explore seulement les sous-espaces d'états critiques par simulation jusqu'à leur détection, ensuite la vérification formelle prend le relais [3] . Cette approche, malheureusement, ne garantit pas une équivalence entre les deux modèles de simulation et de vérification sur lesquelles elle repose. En revanche, le modèle DEv-PROMELA décrivant le même NoC permet l'extraction à la fois d'un modèle PROMELA pour vérifier la propriété d'absence d'interblocage et d'un modèle de simulation DEVS pour quantifier certaines variables de l'interblocage. Rappelons que les modèles extraits préservent les propriétés structurelles et comportementales du modèle initial DEv-PROMELA. Ces différents modèles sont consultables à l'adresse URL `https://github.com/khemiriabdelhak/NetworkOnChip`.

3.3 Comparaison des résultats

Nous proposons de modéliser le NoC 3×3 dont certaines unités de calcul sont défectueuses à l'aide de DEv-PROMELA puis d'utiliser les modèles DEVS et PROMELA ainsi obtenus afin de les combiner et de vérifier que le système ne contient pas d'interblocage. L'approche combinée est comparée à l'approche classique de model checking tel que proposé par l'outil de vérification SPIN. Les deux approches sont comparées au regard du nombre d'états visités, du nombre de transitions effectuées, de la quantité de mémoire utilisée et de la durée totale de la recherche. Dans le cas de l'approche combinée, le temps de simulation pour atteindre l'état critique est indiqué. Les résultats sont résumés dans Table 1.

La vérification classique du modèle a épuisé toutes les ressources sans trouver d'erreur et a atteint les limites mémoire avant d'arrêter la recherche avec les options *collapse compression*, *hash-compact* et *exhaustive*. Avec l'option *exhaustive*, la recherche a été arrêtée avant de conclure après 16.2 secondes et 5 Go de mémoire utilisée, ce qui correspond à $1\ 660\ 720$ états uniques explorés et $3\ 670\ 664$ transitions effectuées. Avec l'option *collapse compression*, la recherche a épuisé toute la mémoire disponible après l'exploration de 83 millions d'états uniques, et plus de 10^8 transitions effectuées le tout en 577 secondes. L'option *hash-compact*, bien que permettant l'exploration de plus d'états (plus de 10^8), n'a pas pu trouver d'erreur, et a interrompu la recherche après 491 secondes. La vérification classique du modèle montre des résultats décourageants par rapport à l'approche combinée qui a trouvé l'erreur d'interblocage. En effet, pour chacune des configurations considérées, l'approche combinée a utilisé moins de 140 Mo de mémoire, effectué 2052 transitions et exploré 1992 états uniques en moins d'une seconde.

	MC Classique	Approche combinée
Exhaustive		
Erreur trouvée	Aucune (interrompu)	Interblocage
Nombre d'états	1 660 720	1 992
Nombre de transitions	3 670 664	2 052
Consommation mémoire (Mo)	5 119.940	135.565
Temps de vérification (s)	16.2	0.02
Temps de simulation (s)	N/A	2.20
Temps total (s)	16.2	2.22
Collapse compression		
Erreur trouvée	Aucune (interrompu)	Interblocage
Nombre d'états	83 104 696	1 992
Nombre de transitions	186 830 090	2 052
Consommation mémoire (Mo)	5 119.831	130.097
Temps de vérification (s)	577	0.01
Temps de simulation (s)	N/A	2.20
Temps total (s)	577	2.21
Hash-compact		
Erreur trouvée	Aucune (interrompu)	Interblocage
Nombre d'états	117 803 750	1 992
Nombre de transitions	152 787 170	2 052
Consommation mémoire (Mo)	5 119.831	129.315
Temps de vérification (s)	491	0.01
Temps de simulation (s)	N/A	2.20
Temps total (s)	491	2.21

TABLE 1 – Comparaison des performances de l'approche combinée avec l'approche classique de Model Checking (MC) pour un NoC 3×3

4 Conclusion

La simulation fournit des mesures quantitatives et qualitatives qui reflètent le comportement du modèle et qui ne peuvent pas être obtenues par une requête du modèle de vérification. Par conséquent, une simulation peut être utilisée pour identifier — au moyen de conditions sur les mesures effectuées — un sous-ensemble d'états où une propriété donnée est plus susceptible d'être insatisfaite. Une simulation peut alors permettre à la procédure de vérification de se concentrer sur un sous-ensemble de l'espace total d'état, et de limiter le problème de l'explosion de cet espace d'états. Cependant, il est nécessaire de garantir une équivalence entre le modèle de simulation et de vérification. Malheureusement, cette équivalence reposait sur les capacités du modélisateur à créer deux modèles, dans deux formalismes, qui soient équivalents au regard de la propriété à vérifier. Pour cela, nous avons montré au travers d'une étude de cas, la pertinence de l'utilisation du formalisme DEv-PROMELA décrivant le même système, et qui permet l'extraction à la fois d'un modèle PROMELA et d'un modèle DEVS qui préservent les propriétés structurelles et comportementales du modèle initial DEv-PROMELA.

Dans nos travaux futurs, nous allons nous intéresser à la mise à l'échelle de cette étude de cas et mettre en lumière les avantages et inconvénients de notre approche combinée reposant sur le langage DEv-PROMELA.

Remercients

Nous remercions les trois relecteurs anonymes de notre article pour les commetaires très constructifs et les présidents du comité de programme d'AFADL de nous avoir donné l'opportunité de publier nos travaux de recherche.

Références

[1] É. Cota, A. de Morais Amory, and M. Soares Lubaszewski. *Reliability, Availability and Serviceability of Networks-on-Chip*. Springer US, 1st edition, 2012.

[2] G. Holzmann. *The SPIN Model Checker : Primer and Reference Manual*. Addison-Wesley Professional, 1st edition, 2011.

[3] A. Khemiri, M. Hamri, C. Frydman, and J. Pinaton. Limiting State Space Explosion of Model Checking Using Discrete Event Simulation : Combining DEVS and PROMELA. In *Proceedings of Computer Simulation Conference 2019*, Berlin, Germany, July 2019.

[4] A. Yacoub. *Une approche de vérification formelle et de simulation pour les systèmes à événements : application à PROMELA*. PhD thesis, 2016. Thèse de doctorat dirigée par Frydman, C. et Hamri, M. Informatique Aix-Marseille 2016.

[5] A. Yacoub, M. Hamri, and C. Frydman. Complementarity between simulation and formal verification transformation of PROMELA models into FDDEVS models : Application to a case study. In *4th International Conference On Simulation And Modeling Methodologies, Technologies And Applications, SIMULTECH 2014, Vienna, Austria, August 28-30, 2014*, pages 421–426. IEEE, 2014.

[6] A. Yacoub, M. Hamri, and C. Frydman. A method for improving the verification and validation of systems by the combined use of simulation and formal methods. In *18th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications, DS-RT 2014, Toulouse, France, October 1-3, 2014*, pages 155–162. IEEE Computer Society, 2014.

[7] A. Yacoub, M. Hamri, and C. Frydman. Using DEv-PROMELA for modelling and verification of software. In *Proceedings of the ACM Conference on SIGSIM Principles of Advanced Discrete Simulation, SIGSIM-PADS 2016, Banff, Alberta, Canada, May 15-18, 2016*, pages 245–253. ACM, 2016.

[8] A. Yacoub, M. Hamri, C. Frydman, C. Seo, and B. P. Zeigler. DEv-PROMELA : an extension of PROMELA for the modelling, simulation and verification of discrete-event systems. *IJSPM*, 12(3/4) :313–327, 2017.

[9] B. P. Zeigler, A. Muzy, and E. Kofman. *Theory of Modeling and Simulation : Discrete Event & Iterative System Computational Foundations*. Academic Press, Inc., USA, 3rd edition, 2018.

Directed Fuzzing for Use-After-Free Vulnerabilities Detection

(*Doctoral Work Presentation*)

Manh-Dung Nguyen [†]
Université Paris-Saclay, CEA LIST, France
manh-dung.nguyen@cea.fr

1 Introduction

Context. *Coverage-based Greybox Fuzzing* (CGF) [1] shows its ability to find various types of bugs in real-world applications. While the main goal of CGF is to cover as many program states as possible in a limited time, *Directed Greybox Fuzzing* (DGF) [2,3] aims to perform stress testing on potentially vulnerable target locations, with applications to different security contexts: (1) bug reproduction, (2) patch testing or (3) static analysis report verification. We focus mainly on *bug reproduction*, which is the most common practical application of DGF. It consists in generating Proof-of-Concept (PoC) inputs of disclosed vulnerabilities given bug report information. It is especially needed since only 54.9% of usual bug reports can be reproduced [4]. Even with a PoC provided in the bug report, developers may still need to consider all corner cases of the bug in order to avoid regression bugs or incomplete fixes. Bug stack traces, sequences of function calls at the time a bug is triggered, are widely used for guiding directed fuzzers [2, 3].

Problems. Fuzzers have made tremendous progress on specific problems (e.g., magic bytes comparison, deep execution, lack of directedness and complex file formats), but existing greybox fuzzers still have a hard time finding *complex vulnerabilities* such as Use-After-Free (UAF), non-interference or flaky bugs, which require bug-triggering paths satisfying very specific properties. *We focus on UAF bugs – one of the most critical exploitable vulnerabilities.* They appear when a heap element is used after having been freed (Listing 1) and have serious consequences.

```
1 char *buf = (char *) malloc(BUF_SIZE);
2 free(buf); // pointer buf becomes dangling
3 ...
4 strncpy(buf, argv[1], BUF_SIZE-1); // Use-After-Free
```

Listing 1: Code snippet illustrating a UAF bug.

[†]PhD student co-supervised at CEA LIST & Univ. Grenoble Alpes (27/11/2017 – 20/11/2020). Thanks to my supervisors for their inputs: Roland Groz, Richard Bonichon, Sébastien Bardin and Matthieu Lemerre. This research was supported by a grant from the H2020 C4IIOT project.

Challenges. Fuzzers targeting the detection of UAF bugs confront themselves with the following challenges.

- **C1. Complexity** – Exercising UAF bugs require to generate inputs triggering a *sequence* of 3 events – *alloc*, *free* and *use* – *on the same memory location*, spanning multiple functions of the program under test (PUT). This combination of both *temporal* and *spatial* constraints is difficult to meet in practice;
- **C2. Silence** – UAF bugs often have *no observable effect*, such as segmentation faults. Thus, fuzzers simply observing crashing behaviors do not detect that a test case triggered such bugs. Sadly, popular profiling tools such as ASan or VALGRIND cannot be used in a fuzzing context due to their high overhead.

Related Work. AFLGO [2] was the first directed fuzzer. It features a simulated annealing-based power schedule that gradually favors seeds whose traces are closer to the target locations. HAWKEYE [3] improves it in terms of seed prioritization, power scheduling and mutation strategies. Although DGF effectively solves the reachability problem of the target locations, existing DGF is limited in detecting UAF vulnerabilities in binaries even given the UAF bug target locations.

Research Directions. We propose 3 research directions as follows:

- **D1.** *Directed fuzzer towards* UAF *targets extracted from a bug trace*, which is a sequence of function calls at the time a bug is triggered (*discussed in details in this paper*). It could be obtained by running the PUT with a PoC input under profiling tools such as VALGRIND [5].
- **D2.** *Typestate directed fuzzer detects common bugs* (e.g., UAF, buffer overflows) that are considered as the violation of typesate properties (*work in progress*).
- **D3.** *Hybrid directed fuzzer towards* UAF *targets extracted from static reports* of UAF detector like GUEB [6] (*as future work*). GUEB, which is the *only binary-level static analyzer for UAF*, performs a dedicated value-set analysis with different heap modeling approaches.

Contributions. Our work **D1** [7, 8] makes the followings contributions:

- We design the *first* directed greybox fuzzing technique dedicated to *UAF bugs* working directly *on executables*.
- We implement the fuzzer UAFUZZ on top of AFL [1] and BINSEC [9].
- We construct a *new UAF fuzzing benchmark* [10] of 30 real UAF bugs.
- For *bug reproduction* setting, we evaluate UAFUZZ against state-of-the-art coverage-guided and directed greybox fuzzers using our UAF fuzzing benchmark. For *patch testing* setting, UAFUZZ successfully discovers 11 new UAF (4 buggy patches) in critical projects like Perl, GNU Patch and GPAC.

2 The UAFUZZ approach

Overall we propose 3 dynamic input metrics specialized for UAF vulnerabilities detection, used in conjunction with a dedicated power schedule assigning more energy (i.e., number of mutants) to favored seeds that are wisely selected by our new heuristic during fuzzing (as shown in Fig. 1).

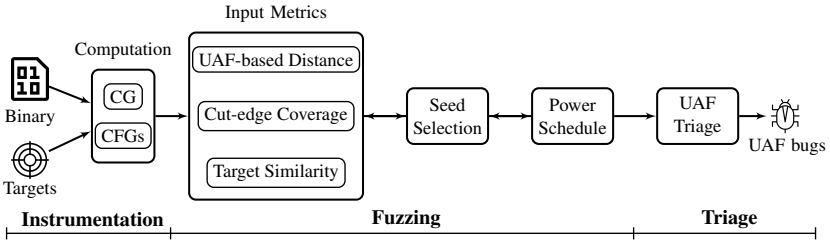


Figure 1: The UAFUZZ workflow.

Target Similarity metric & Seed Selection heuristic. A *target similarity metric* measures the similarity between the execution of a seed and the target UAF bug trace. Our seed selection algorithm is based on two insights. First, we *should prioritize seeds that are most similar to the target bug trace*. Second, *target similarity should take ordering (a.k.a., sequenceness) into account*, as traces covering sequentially a number of locations in the target bug trace are closer to the target than traces covering the same locations in an arbitrary order.

UAF-based Distance metric. Previous seed distances [2, 3] do not account for any order among the target locations, while it is essential for UAF. We address this issue by modifying the distance between functions in the call graph to favor paths that *sequentially* go through the 3 UAF events *alloc, free and use* of the bug trace.

Cut-edge Coverage metric. We propose the *lightweight* cut-edge coverage metric by measuring the "*reachability progress*" at the *edge level* but on the *critical decision nodes only*. Our heuristic is that an input exercising more cut edges whose edge destinations are more likely to reach the next target in the bug trace, is more likely to cover more locations of the target UAF bug trace.

Power Schedule. The power scheduler determines the energy for each selected seed based on its dynamic metric scores at runtime. We therefore spend more time fuzzing seeds that (1) *are closer* (using the UAF-based seed distance); (2) are *more similar to the UAF target bug trace* (using the target similarity); (3) *make better decisions* at critical code junctions (using the cut-edge coverage).

UAF Triage. False positive inputs are finally filtered by running the PUT with *potential inputs* which exercise in sequence all target locations of the target UAF bug trace under a profiling tool (here VALGRIND [5]) in the bug triaging phase.

3 Current Results

Implementation. We develop a lightweight static analysis as a plugin of the binary analysis platform BINSEC [9] and the dynamic fuzzing part based on AFL-QEMU 2.52b [1]. We also re-implement the best state-of-the-art DGF, named AFLGOB[1]

[1]The comparison between AFLGOB and source-based AFLGO is discussed in [7].

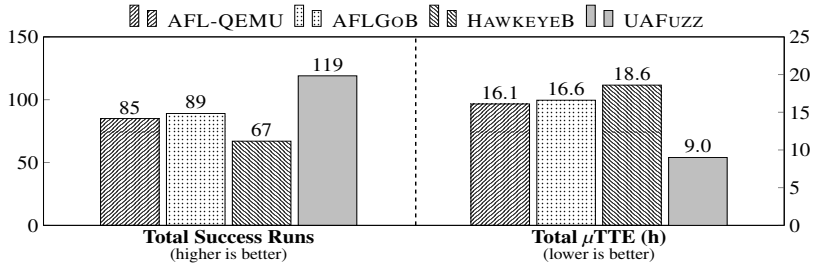


Figure 2: Summary of fuzzing performance (RQ1)

and HAWKEYEB, in our own framework for binary-level fuzzing because HAWKEYE [3] is not available and AFLGO [2] works on source code only.

RQ1 – Bug-reproduction Ability. Overall UAFUZZ *significantly outperforms* existing fuzzers in terms of UAF bugs reproduction. We use Time-to-Exposure (TTE) and the number of success runs in which a fuzzer could trigger the bug as two important metrics to compare the efficiency of each fuzzer. As shown in Fig. 2, UAFUZZ has the *largest* total number of success runs and achieves 34% (up to $+300\%$) more than the second best fuzzer AFLGOB. Furthermore, UAFUZZ can find bugs around $2\times$ faster than other fuzzers. Interestingly, UAFUZZ is able to find the bugs faster than AFLGO with source code in 4 cases, which implies the efficiency of our techniques.

RQ2 – Overhead. UAFUZZ is *an order of magnitude faster* ($14.7\times$ in total) than the source-based directed fuzzer AFLGO in the instrumentation phase. The total numbers of executions done of UAFUZZ are *slightly smaller* (-4% in total) than AFL-QEMU, which implies that UAFUZZ's runtime overhead is negligible.

RQ3 – UAF Triage. UAFUZZ reduces a *large portion* (i.e., more than 90% of triaging inputs) in the post-processing phase. Unsurprisingly, UAFUZZ spends the *smallest* amount of time (i.e., an average of $7.4s$ – a speedup of up to $17.5\times$ over AFLGOB) in the bug triaging step thanks to our target similarity metric.

RQ4 – Individual Contribution. Our experiments show that UAF-based distance, power schedule and seed selection heuristic *individually contribute* to improve the performance of corresponding variants built on top of AFLGOB. Combining these components can *further improve* effectiveness and efficency of our technique.

4 Conclusion & Future Work

UAFUZZ is the *first directed* greybox fuzzing approach tailored to detecting UAF bugs *in binary* given only the bug trace. By specializing standard (directed) greybox fuzzing components to UAF, UAFUZZ outperforms existing directed fuzzers, both in terms of time to bug exposure and number of success runs. Our technique also enjoys only a small overhead (instrumentation- and run- time), and speeds up the bug triage step by significantly reducing the number of seeds to be sent to an external UAF checker. Finally, we currently follow the directions **D2** and **D3**.

References

[1] "Afl," http://lcamtuf.coredump.cx/afl/, 2020.

[2] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS17)*, 2017.

[3] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu, "Hawkeye: towards a desired directed grey-box fuzzer," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 2095–2108.

[4] D. Mu, A. Cuevas, L. Yang, H. Hu, X. Xing, B. Mao, and G. Wang, "Understanding the reproducibility of crowd-reported security vulnerabilities," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, 2018, pp. 919–936. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/mu

[5] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *ACM Sigplan*, vol. 42-6. ACM, 2007, pp. 89–100.

[6] J. Feist, L. Mounier, and M.-L. Potet, "Statically detecting use after free on binary code," *Journal of Computer Virology and Hacking Techniques*, vol. 10, no. 3, pp. 211–217, 2014.

[7] M.-D. Nguyen, S. Bardin, R. Bonichon, R. Groz, and M. Lemerre, "Binary-level directed fuzzing for use-after-free vulnerabilities," *The 23nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID '20)*, 2020.

[8] ——, "About Directed Fuzzing and Use-After-Free: How to Find Complex & Silent Bugs?" Black Hat USA, 2020.

[9] "Binsec," https://binsec.github.io/, 2020.

[10] "Uaf fuzzing benchmark," https://github.com/uafuzz/UAF-FuzzBench, 2020.

BINSEC/REL : Exécution Symbolique Relationnelle Efficace pour l'Analyse de Binaire Constant-Time

Lesly-Ann Daniel[1], Sébastien Bardin[1], and Tamara Rezk[2]

[1]CEA List, Université Paris-Saclay
[2]INRIA Sophia-Antipolis, INDES Project
lesly-ann.daniel@cea.fr, sebastien.bardin@cea.fr, tamara.rezk@inria.fr

Résumé

Constant-time est une contre-mesure aux attaques temporelles qui interdit les branchements et les accès mémoires dépendants des secrets. Cette contre-mesure n'est généralement pas préservée par le compilateur et requiert donc de raisonner au niveau binaire. Or, les outils d'analyse dédiés à constant-time raisonnent actuellement à un plus haut niveau (C ou LLVM), approximent la sémantique du programme, ou ne passent pas à l'échelle. Nous concevons une technique d'analyse efficace au niveau binaire qui ne fait pas d'approximation sur la sémantique du programme, permettant à la fois de trouver des bugs ou de faire de la vérification bornée pour constant-time. Celle-ci s'appuie sur l'exécution symbolique relationnelle, à laquelle nous ajoutons des optimisations dédiées. Nous proposons un prototype, BINSEC/REL et réalisons des expériences sur un ensemble de 338 binaires cryptographiques, démontrant le passage à l'échelle de notre technique. De plus, en utilisant BINSEC/REL, nous avons automatisé et étendu une étude existante sur la préservation de constant-time par les compilateurs. Nous avons ainsi découvert des violations introduites par les compilateurs qui étaient hors de portée des outils d'analyse pour LLVM, soulignant l'importance de raisonner au niveau binaire.

Objet : AFADL 2020 résumé long. Travail original accepté à S&P 2020 [1].

Contexte : Constant-time contre les attaque temporelles.
Les attaques temporelles permettent à un attaquant capable de surveiller le temps d'exécution d'un système, d'extraire des informations sur les secrets manipulés par un programme (e.g. clés cryptographiques). Ces attaques exploitent des corrélations entre les secrets et le temps d'exécution du programme qui peuvent subvenir quand le *flot de contrôle* du programme dépend des secrets, mais aussi à travers ses *accès mémoire*, via les attaques par cache. La discipline de programmation *constant-time* est une contre-mesure qui permet de décorréler le temps d'exécution des secrets manipulés par un programme. Un programme est constant-time si

pour chaque traces d'exécution t et t' avec les même entrées publiques, t et t' ont le même flot de contrôle et les mêmes accès mémoire, peu importe la valeur des entrées secrètes. Cette contre-mesure est aujourd'hui largement utilisée pour sécuriser les implémentations cryptographiques (e.g. BearSSL, NaCL, HACL*, etc).

Problème : Un outil efficace pour vérifier constant-time au niveau binaire ?
Écrire une implémentation constant-time n'est pas trivial et requiert l'utilisation d'opérations bas niveau pour remplacer l'utilisation de branchements conditionnels. De plus, constant-time n'est généralement pas préservé par le compilateur [2]. Par exemple, le code `c=(x<y)-1` peut être, ou non, compilé vers un saut conditionnel selon la version du compilateur et les optimisations utilisées.

Plusieurs outils d'analyse de constant-time ciblent le code source [3] ou le bytecode LLVM [4], laissant la porte ouverte aux vulnérabilités introduites par le compilateur. Les outils d'analyse pour constant-time ciblant le binaire se basent soit sur des approches dynamiques [5], [6] qui peuvent trouver des bugs mais sont incomplètes ; soit sur des approches statiques [7] qui peuvent garantir qu'un programme est constant-time mais ne peuvent reporter de bugs précis.

Il manque donc un outil efficace d'analyse de constant-time au niveau binaire qui puisse à la fois trouver des bugs ou garantir leur absence.

Défis de la vérification de constant-time.

- Les outils de vérification standards ne s'appliquent pas directement car constant-time est une propriété de 2-hypersûreté [8] (i.e. reliant deux exécutions). Ces propriétés peuvent être réduites à des propriétés de sûreté sur un programme transformé via *auto-composition* [9], mais cette réduction est inefficace [10].
- Par ailleurs, il est connu que l'adaptation des méthodes formelles à l'analyse de binaire est complexe, notamment à cause de la perte de structure au niveau des données et du flot de contrôle et de la représentation explicite de la mémoire sous forme d'un large tableau d'octets.

L'exécution symbolique pour la 2-hypersûreté.
L'*exécution symbolique* (SE) [11] est une technique d'analyse qui permet à la fois de trouver des bugs ou de faire de la vérification bornée et qui a fait ses preuves en analyse de binaire. En revanche, les tentatives d'adaptation de cette dernière à des propriétés de 2-hypersûreté ne passent pas à l'échelle [12].

Une idée émergente consiste à représenter *deux traces d'exécution dans la même instance d'exécution symbolique* en maximisant le partage entre ces deux exécutions. Cette idée fut introduite sous le terme *ShadowSE* [13] dans le contexte du test de version, puis reprise dans le contexte de la vérification de 2-hypersûreté sous le terme *exécution symbolique relationnelle* [14].

L'exécution symbolique relationnelle.
L'exécution symbolique relationnelle (RelSE) [14] permet de modéliser deux traces d'exécution dans la même instance d'exécution symbolique en associant chaque variable soit à une *paire d'expressions symboliques* $\langle \varphi \,|\, \varphi' \rangle$ quand celle-ci *peut dépendre* des secrets ou alors à une *expression simple* $\langle \varphi \rangle$ quand celle-ci *ne dépend*

pas des secrets. Pour analyser constant-time, une requête est envoyée au solveur à chaque branchement ou accès mémoire dépendant d'une paire d'expressions afin de s'assurer que celle-ci ne dépend pas des secrets. En revanche, dans le cas d'une expression simple, l'appel au solveur n'est pas nécessaire puisque, par définition, celle-ci ne dépend pas des secrets. RelSE permet ainsi de partager les expressions similaires dans les deux exécutions et de réduire les appels au solveur en traçant les dépendances aux secrets.

Malheureusement, l'adaptation directe de RelSE ne passe pas à l'échelle dans le contexte de l'analyse de binaire. La mémoire, représentée sous forme d'un large tableau d'octets, est dupliqué dès le début de l'exécution symbolique, empêchant le partage entre les deux exécutions et le traçage des dépendances aux secrets.

Proposition. Nous proposons une technique basée sur l'exécution symbolique relationnelle qui permet de trouver des bugs et de faire de la vérification bornée efficace de constant-time au niveau binaire.

Contributions. Nos contributions sont les suivantes :

- Nous concevons trois *optimisations* pour RelSE dédiées à l'analyse de constant-time au niveau binaire : (1) *FlyRow*, variante à la volée du *read-over-write* [15], permet d'éviter de recourir à la mémoire dupliquée en résolvant les lectures en avant du solveur. (2) *Untainting* permet de transformer les paires d'expressions jugées égales par le solveur en expressions simples, favorisant le partage entre les exécutions. (3) *Fault-packing* permet de regrouper les requêtes pour un même bloc de base afin de limiter le nombre d'appels au solveur.

  De plus, nous prouvons que notre analyse est correcte pour la recherche de bugs et la vérification bornée de constant-time. Plus précisément, si l'analyse *trouve un bug* alors il existe deux traces d'exécution du programme qui, avec les mêmes entrées publiques, produisent un flot de contrôle ou des accès mémoire différents. Par ailleurs, si l'analyse *ne trouve pas de bug* jusqu'à une certaine borne, alors le programme est constant-time jusqu'à cette borne. Dans le cadre de primitives cryptographiques, cela revient à effectuer la vérification pour une taille d'entrée bornée.

- Nous proposons BINSEC/REL, le premier outil d'analyse automatique efficace pour la recherche de bugs et la vérification bornée de constant-time. Il est agnostique quant au compilateur, peut être utilisé pour des architectures x86 et ARM et ne requiert pas le code source. BINSEC/REL peut analyser 23 millions d'instructions en 98 minutes (3860 instructions par seconde) [1] tout en étant correct et complet pour constant-time.

  Nous l'évaluons sur en échantillon de 338 binaires cryptographiques et montrons qu'il peut trouver des bugs ou vérifier une implémentation 700 fois

1. Évaluation réalisée sur un ordinateur portable avec un processeur Intel(R) Core(TM) i5-2520M CPU @ 2.50GHz et 32GB de RAM, avec Linux Mint 18.3 Sylvia

plus vite que RelSE, rendant possible l'analyse de vraies primitives cryptographiques.

- Parmi notre échantillon de 338 binaires cryptographiques, 296 respectent constant-time et les 42 autres comportent des bugs (connus).

  Nous donnons de nouvelles preuves bornées de constant-time au niveau binaire pour les 296 programmes constant-time, précédemment vérifiés à plus haut-niveau (e.g. HACL*, BearSSL, NaCL). Plus précisément, pour une taille d'entrée donnée, BINSEC/REL est capable d'explorer exhaustivement ces programmes sans trouver de violations.

  Pour les 42 autres programmes, BINSEC/REL est capable de retrouver les bugs connus (e.g. Lucky13) et de fournir des contre-exemples.

- Nous étendons une étude existante sur la préservation de constant-time par le compilateur `clang` [2] pour des binarires x86. Nous passons d'une analyse manuelle à une analyse automatique. Nous l'étendons pour prendre en compte 29 nouvelles fonctions, le compilateur `gcc`, une nouvelle versions de `clang` ainsi que des binaires ARM.

  Nous avons découvert que `gcc -O0` et les passes finales de `clang` peuvent introduire des vulnérabilités hors de portée des outils de vérification pour LLVM comme `ct-verif` [4]. Ce dernier point souligne l'importance de développer des outils de vérification pour constant-time au niveau binaire.

Références

[1] L.-A. DANIEL, S. BARDIN et T. REZK, "Binsec/Rel: Efficient Relational Symbolic Execution for Constant-Time at Binary-Level", in *2020 IEEE Symposium on Security and Privacy (SP)*.

[2] L. SIMON, D. CHISNALL et R. J. ANDERSON, "What You Get is What You C: Controlling Side Effects in Mainstream C Compilers", in *EuroS&P*, 2018.

[3] S. BLAZY, D. PICHARDIE et A. TRIEU, "Verifying Constant-Time Implementations by Abstract Interpretation", in *ESORICS*, 2017.

[4] J. B. ALMEIDA, M. BARBOSA, G. BARTHE, F. DUPRESSOIR et M. EMMI, "Verifying Constant-Time Implementations.", in *USENIX*, 2016.

[5] S. WANG, P. WANG, X. LIU, D. ZHANG et D. WU, "CacheD: Identifying Cache-Based Timing Channels in Production Software", in *USENIX*, 2017.

[6] J. WICHELMANN, A. MOGHIMI, T. EISENBARTH et B. SUNAR, "MicroWalk: A Framework for Finding Side Channels in Binaries", in *ACSAC*, 2018.

[7] G. DOYCHEV, B. KÖPF, L. MAUBORGNE et J. REINEKE, "CacheAudit: A Tool for the Static Analysis of Cache Side Channels", *ACM Transactions on Information and System Security*, t. 18, n° 1, 2015.

[8] M. R. CLARKSON et F. B. SCHNEIDER, “Hyperproperties”, *Journal of Computer Security*, t. 18, n° 6, 2010.

[9] G. BARTHE, P. R. D’ARGENIO et T. REZK, “Secure Information Flow by Self-Composition”, in *CSFW*, 2004.

[10] T. TERAUCHI et A. AIKEN, “Secure information flow as a safety problem”, in *SAS*, 2005.

[11] J. C. KING, “Symbolic execution and program testing”, *Commun. ACM*, t. 19, n° 7, 1976.

[12] D. MILUSHEV, W. BECK et D. CLARKE, “Noninterference via symbolic execution”, in *Formal Techniques for Distributed Systems*, 2012.

[13] H. PALIKAREVA, T. KUCHTA et C. CADAR, “Shadow of a doubt: testing for divergences between software versions”, in *ICSE*, 2016.

[14] G. P. FARINA, S. CHONG et M. GABOARDI, “Relational Symbolic Execution”, in *PPDP*, 2019.

[15] B. FARINIER, R. DAVID, S. BARDIN et M. LEMERRE, “Arrays Made Simpler: An Efficient, Scalable and Thorough Preprocessing”, in *LPAR*, 2018.

Vers la vérification d'une méthodologie pour la conception de circuits numériques critiques

Vincent Iampietro[1], David Andreu[1,2], and David Delahaye[1]

[1]LIRMM, Université de Montpellier, CNRS, Montpellier, France
[2]NEURINNOV, Montpellier, France

1 Introduction

Pour répondre aux contraintes liées à la conception de circuits numériques critiques, et à l'augmentation constante de la complexité des systèmes, le domaine de l'Ingénierie Système à Base de Modèles (ISBM) a été développé. L'intérêt est de travailler sur des modèles de haut niveau avec un pouvoir d'expression et des qualités de compréhension et de lisibilité qui facilitent les interactions entre les acteurs de la conception du circuit (i.e, les ingénieurs). Plusieurs formalismes existent : le langage SysML [2], des variantes du langage C [9], ou encore les réseaux de Petri (RdPs) [8], pour citer les plus répandus. Une fois la conception terminée, les modèles sont physiquement synthétisés en suivant un procédé manuel ou automatique. Il reste alors à prouver que la phase de transformation préserve le comportement du modèle de conception. Le présent article décrit le travail d'une thèse en cours. Cette thèse s'intéresse à la vérification d'un processus d'aide à la modélisation et à la production de circuits numériques critiques : la méthodologie HILECOP (HIgh LEvel hardware COmponent Programming). Cette méthodologie est mise en oeuvre dans le cadre de la création de micro-contrôleurs intégrés à des dispositifs médicaux de type neuroprothèses. La Figure 1 en décrit les principales étapes.

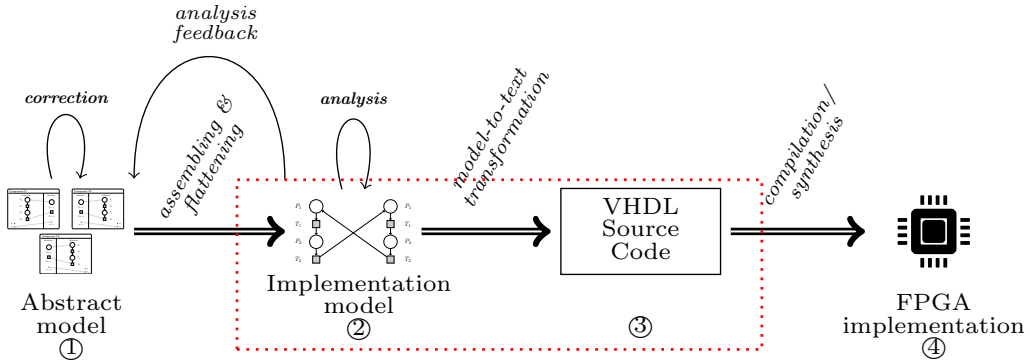


Figure 1 – La méthodologie HILECOP

Le concepteur de systèmes électroniques esquisse premièrement un modèle graphique de haut niveau de son circuit (①). Ce modèle s'appuie sur le formalisme des diagrammes à composants, avec l'addition des RdPs pour décrire le comportement interne des parties du circuit. Dans un deuxième temps, les parties du modèle sont assemblées et la structure des composants est effacée. Le résultat obtenu est un réseau de Petri global décrivant le système modélisé (②). Des outils d'analyse exploités par la méthodologie permettent alors de vérifier certaines propriétés du modèle (caractère borné, vivacité…) et présentent un compte-rendu au concepteur. Après plusieurs itérations du cycle analyse-correction, du code VHDL est généré à partir du modèle d'implémentation (③). Dès lors, la dernière étape de la méthodologie, qui opère la synthèse du circuit électronique depuis le code source VHDL, est prise en charge par un compilateur/synthétiseur industriel propriétaire (④).

L'objectif de la thèse est de prouver que la transformation du modèle d'implémentation en code VHDL (i.e, de ② vers ③ dans la Figure 1) n'introduit pas de divergences de comportement. Dans cette optique, il sera nécessaire de formaliser la sémantique des modèles de haut niveau (RdP), du langage cible (VHDL), et de décrire la transformation. Ensuite, la preuve de similarité comportementale devra être établie. L'intégralité de la démarche sera mécanisée avec l'assistant à la preuve Coq [6]. Même si cette démarche a été éprouvée pour la vérification de compilateurs,

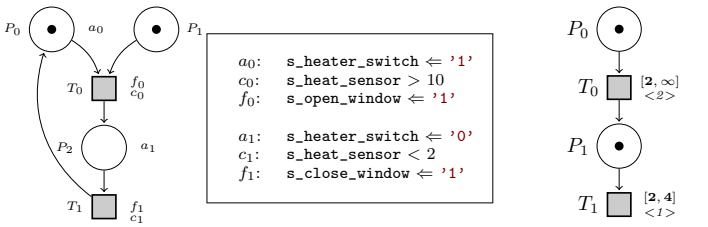
son application à la conception de circuits numériques est bien moins fréquente. L'intérêt scientifique provient de la distance qui existe entre le modèle d'exécution du formalisme source (SITPN) et celui du langage cible (VHDL). Cette distance devra être prise en compte lors de la preuve de préservation de comportement.

2 Un formalisme de haut-niveau : les réseaux de Petri

Du fait de leur statut de modèles formels et des possibilités d'analyse qui en résultent, les RdPs ont été retenus comme modèles de haut niveau de la méthodologie HILECOP. Le but de la méthodologie étant la conception et la production de circuits numériques *critiques*, les modèles se doivent d'être validés par analyse formelle. Afin d'augmenter l'expressivité des modèles, les RdPs HILECOP combinent plusieurs classes connues de RdPs (présentées ci-après), mais leur particularité réside dans leur exécution synchrone. Les RdPs HILECOP sont nommés SITPNs pour Synchronously executed Interpreted Time Petri Nets with priorities.

Les SITPNs sont des RdP interprétés; des actions peuvent être associées aux places d'un réseau, et des fonctions/conditions peuvent être associées aux transitions. Actions et fonctions définissent des opérations qui influencent l'environnement du RdP, et leurs effets sont mesurés à travers la valeur booléenne des conditions associées aux transitions. Dans un RdP interprété, une transition est franchissable si elle est sensibilisée et que toutes les conditions qui lui sont associées sont *vraies*. La Figure 2.(a) donne un exemple de RdP interprété où les actions, fonctions et conditions sont définies par du code VHDL.

Les RdP utilisés dans HILECOP sont temporels; une fenêtre de tir, i.e un intervalle de temps, peut être associée à une transition. Un compteur de temps est lancé lorsqu'une transition devient sensibilisée; celle-ci devient franchissable lorsque son compteur de temps a atteint l'intervalle de tir. La Figure 2.(b) donne un exemple de RdP temporel. La valeur courante des compteurs de temps est représentée entre chevrons en dessous des intervalles temporels associés. En résumé, une transition d'un SITPN est franchissable si elle est sensibilisée, si toutes les conditions qui lui sont associées sont vraies et si son compteur de temps est dans l'intervalle défini.



(a) RdP interprété.

(b) RdP temporel.

Figure 2 – Réseaux de Petri temporels et interprétés.

Contrairement au cas général, les SITPNs ont une politique de tir (i.e, une sémantique) *synchrone*. Fondamentalement, le tir des transitions d'un RdP est un phénomène indéterministe (si deux transitions sont franchissables au même instant, tous les ordres de tirs sont possibles), et asynchrone (dès qu'une transition est franchissable, elle peut être tirée sans attente). A contrario, l'évolution d'un SITPN est rythmée par le front montant et le front descendant d'un signal d'horloge, comme montré dans la Figure 3. Sur le front descendant (① de la Figure 3), toutes les transitions devant être tirées sont déterminées, ce après mise à jour des conditions et intervalles de temps; sur le front montant (② de la Figure 3), les précédentes transitions sont tirées, entraînant la mis à jour du marquage du réseau et l'exécution de fonctions. La sémantique d'évolution d'un tel réseau est synchrone et déterministe.

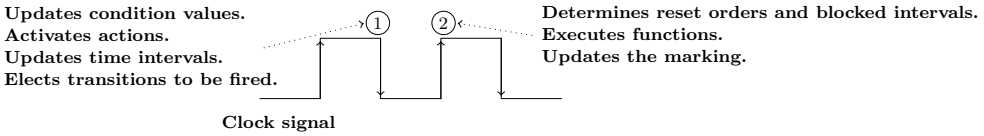


Figure 3 – Evolution synchrone d'un SITPN.

La structure et la sémantique des SITPNs ont été formalisées dans [4, 5]. La sémantique est exprimée comme un système états-transitions où les transitions sont étiquetées par les évènements d'un signal d'horloge. Il y a deux évènements possibles : le front montant et le front descendant du signal. L'état d'un SITPN décrit, entre autres, le marquage courant du SITPN, la valeur des compteurs de temps et des conditions associés aux transitions, la liste des transitions à tirer…La sémantique des SITPNs fixe les règles de changement d'état en fonction des évènements d'horloge. Par exemple, sur le front descendant d'horloge, la liste des transitions à tirer au prochain front montant est calculée; une règle stipule qu'une aucune transition non franchissable au front descendant n'appartient à l'ensemble des transitions à tirer.

Algorithme 1 : SimulationLoop(Δ, σ_{init}, $nbCycles$)

```
1  begin
       // Initialization phase.
2      σ1 = RunAllProcessesOnce(Δ, σinit)
3      σ2 = Stabilize(Δ, σ1)
       // Main loop.
4      Tc ← 0
5      while Tc ≤ nbCycles do
6          σ3 = ExecuteFallingEdgePss(Δ, σ2)
7          σ4 = Stabilize(Δ, σ3)
8          σ5 = ExecuteRisingEdgePss(Δ, σ4)
9          σ2 = Stabilize(Δ, σ5)
10         Tc ← Tc + 1
```

La première contribution de la thèse est l'implantation en Coq de la structure et de la sémantique des SITPNs. La sémantique a été implantée comme une relation inductive paramétrée par un SITPN, deux états (i.e, avant et après transition), et un évènement d'horloge. La relation présente deux cas de construction, un pour chaque évènement d'horloge considéré. Afin de tester notre implantation de la sémantique des SITPNs, un interprète a été conçu, i.e un programme qui simule les changements d'état d'un SITPN pour n cycles d'horloge, en partant de l'état initial du réseau. Cet interprète est prouvé correct et complet vis à vis de la sémantique des SITPNs pour une évolution sur un cycle d'horloge. L'intégralité de la formalisation et de la mécanisation est mise à disposition du lecteur[1].

3 Un langage cible : VHDL

Il existe plusieurs techniques permettant la synthèse physique d'un RdP. Cependant, la technique la plus étudiée est la transformation vers la langage VHDL. Cette technique a donc été retenue par la méthodologie HILECOP. Le langage VHDL permet les descriptions structurelle et comportementale de circuits électroniques, à des fins de simulation ou de synthèse physique. En VHDL, un *design* décrit un composant électronique en termes d'interface entrée-sortie (l'*entité*) et de comportement interne (l'*architecture*). Le comportement d'un design s'exprime de deux manières : via l'interconnexion d'instances d'autres designs (des sous-composants), ou à l'aide de *processus*. La spécificité du langage VHDL tient à l'exécution concurrente des processus et des sous-composants décrivant une architecture de design. Un processus définit un bloc d'instructions séquentielles; il observe un certain nombre de signaux qui composent sa liste de sensibilité. Le changement d'état d'un signal de cette liste entraîne l'exécution du bloc d'instructions du processus. Conceptuellement, un signal VHDL représente une connexion physique sur un circuit électronique. Les signaux sont les principaux véhicules des valeurs dans les programmes VHDL.

La sémantique de VHDL est décrite dans une prose informelle dans le manuel de référence du langage (MRL). De fait, interpréter un programme VHDL, qui décrit un *design* de circuit, revient à simuler le design décrit. Dans le MRL, la sémantique de VHDL est donc définie sous la forme d'une boucle de simulation. La boucle de simulation spécifie la dynamique d'exécution des blocs concurrents qui composent une architecture de design, ainsi que la propagation des valeurs au travers des signaux.

La littérature propose de nombreuses formalisations de la sémantique de VHDL [3]. Certaines formalisations expriment la boucle de simulation telle qu'exhibée dans le MRL; d'autres choisissent de s'abstraire de cette boucle, et optent pour une formalisation alternative basée sur des modèles permettant la gestion de la concurrence et du temps (automates temporels, réseaux de Petri, logique d'intervalles temporels...).

La méthodologie HILECOP opère la génération d'un design VHDL dans l'optique de sa synthèse physique. Dès lors, nous ne considérons qu'une partie *synthétisable* du langage que nous définissons et nommons $\mathcal{H}$-VHDL. De plus, les designs VHDL générés par la méthodologie HILECOP décrivent des circuits synchrones, i.e, dont l'exécution est rythmée par un signal d'horloge. La prise en compte d'une sous-partie synthétisable et du synchronisme nous a permis d'exprimer la sémantique des programmes $\mathcal{H}$-VHDL en termes d'une boucle de simulation bien plus simple en comparaison de celle exprimée dans le MRL. L'Algorithme 1 décrit notre boucle spécifique de simulation pour un design $\mathcal{H}$-VHDL.

La boucle de simulation de l'Algorithme 1 est paramétrée par un design VHDL (Δ), l'état initial du design (σ_{init}) qui contient les valeurs des signaux et les états courants des sous-composants du design Δ, et le nombre de cycles de simulation à effectuer ($nbCycles$). Durant la phase d'initialisation, tous les processus décrivant le

[1] https://github.com/viampietro/sitpns

comportement du design sont exécutés une fois. Cette phase est suivie d'une phase de stabilisation de la valeur des signaux. La phase de stabilisation correspond à la propagation des valeurs entre signaux interconnectés, ce jusqu'à ce que la propagation n'induisent plus aucun changement. La boucle principale de simulation décrit l'alternance entre l'exécution des processus dits *séquentiels*, i.e qui sont sensibles aux évènements d'horloge, et des processus *combinatoires*, qui s'exécutent de manière continue jusqu'à stabilisation des signaux.

Au stade actuel des travaux, une formalisation de la sémantique de $\mathcal{H}$-VHDL a été effectuée sous la forme d'une sémantique opérationnelle à grands pas, et sa mécanisation en Coq a été réalisée. Cette sémantique s'inspire des travaux de formalisation esquissés dans [7, 1]. La sémantique formalisée prend également en compte la phase d'élaboration du design, préliminaire à la simulation. L'élaboration génère l'environnement de simulation, i.e un couple Δ, σ_{init} qui se trouve en paramètre de la boucle de simulation (voir Algorithme 1). Durant la phase d'élaboration, une vérification de type est effectué sur le code VHDL. La vérification de type s'assure que la partie déclarative et la partie comportementale du design VHDL respectent certaines règles de typage définies par le MRL. Par exemple, pour une instruction d'affectation de valeur à un signal, l'expression affectée doit être du même type que le signal cible.

4 Conclusion

Le but de la thèse est de vérifier formellement une partie de la méthodologie HILECOP, usitée dans le cadre de la conception de circuits numériques critiques. Spécifiquement, le travail de vérification porte sur la phase transformant un modèle de conception, à base de RdPs, en *design* VHDL. La finalité de ce travail sera la spécification et la démonstration d'un théorème de préservation de comportement pour cette phase de transformation.

Jusqu'ici, la sémantique des SITPNs, modèles de haut niveau de HILECOP, et la sémantique de $\mathcal{H}$-VHDL ont été implantées à l'aide du langage Coq. Concernant les SITPNs, deux éléments déjà existant dans ce formalisme restent à prendre en compte : les macroplaces, qui permettent d'exprimer la gestion d'exceptions dans les SITPNs, ainsi que la possibilité de spécifier des domaines d'horloge différents au sein d'un même modèle; c'est le cas des systèmes Globalement Asynchrones Localement Synchrones (GALS).

La transformation d'un SITPN en un modèle VHDL est en cours de programmation avec le langage Coq. Enfin, la dernière étape de ce travail sera d'établir la preuve de préservation de comportement.

Références

[1] D. Borrione and A. Salem. Denotational semantics of a synchronous VHDL subset. *Formal Methods in System Design*, 7(1–2) :53–71, Aug 1995.

[2] S. Friedenthal, A. Moore, and R. Steiner. *A Practical Guide to SysML : The Systems Modeling Language*. Morgan Kaufmann, Oct 2014.

[3] C. D. Kloos and P. Breuer. *Formal Semantics for VHDL*, volume 307. Springer Science & Business Media, 2012.

[4] H. Leroux. *Méthodologie de conception d'architectures numériques complexes : du formalisme à l'implémentation en passant par l'analyse, préservation de la conformité. Application aux neuroprothèses*. PhD thesis, Université Montpellier II - Sciences et Techniques du Languedoc, Oct 2014.

[5] I. Merzoug. *Validation formelle des systèmes numériques critiques : génération de l'espace d'états de réseaux de Petri exécutés en synchrone*. PhD thesis, Université Montpellier, Jan 2018.

[6] The Coq Development Team. *Coq, version 8.9.0*. Inria, Jan. 2019. `http://coq.inria.fr/`.

[7] J. P. Van Tassel. *An Operational Semantics for a Subset of VHDL*, volume 307, page 71–106. Springer US, 1995.

[8] A. Yakovlev and A. Koelmans. *Petri nets and digital hardware design*, page 154–236. Apr 2006.

[9] Y. Yankova, K. Bertels, S. Vassiliadis, R. Meeuws, and A. Virginia. Automated hdl generation : Comparative evaluation. In *2007 IEEE International Symposium on Circuits and Systems*, page 2750–2753, May 2007.

Vers la vérification de SMALA, un langage réactif interactif

Nicolas Nalpon, Celia Picard, Cyril Allignol et Sébastien Leriche

ENAC, Université de Toulouse - France

Résumé

SMALA est un langage réactif dédié à la programmation de systèmes à forte composante interactive. Pour pouvoir utiliser ce langage dans un contexte critique, il est nécessaire de pouvoir apporter des garanties. Ainsi, nous nous intéressons à la vérification formelle du compilateur de SMALA, avec pour objectif de garantir la préservation de la sémantique du programme source à la compilation. Dans un premier temps, nous avons limité notre étude à un sous-ensemble de SMALA. Dans cet article, nous présentons une version préliminaire de la sémantique opérationnelle de ce sous-ensemble.

Mots-clés : compilation certifiée, langage réactif-interactif, sémantique opérationnelle

1 Introduction

Bon nombre de systèmes critiques comportent une importante composante interactive, c'est-à-dire que des utilisateurs manipulent le système qui doit pouvoir réagir à leurs actions et afficher des informations. C'est notamment le cas de nombreux systèmes aéronautiques, tels que les cockpits d'avions, d'hélicoptères ou les systèmes de contrôle aérien.

Un système qui réagit à des actions issues de son environnement est dit réactif. La programmation réactive est un paradigme qui s'intéresse à la gestion des flux de données dans ces systèmes [2]. Néanmoins, les langages de programmation classiques, même réactifs ou event-driven, sont peu adaptés à la conception de logiciels interactifs. En effet, ils ne permettent pas une intégration fluide et unifiée des différentes phases de conception (design graphique, architecture, codage...) et plus particulièrement de la partie interactive. C'est cette unification des paradigmes réactif et interactif que vise SMALA [5]. Ce langage, dédié à la conception de programmes interactifs, en particulier aéronautiques, a ainsi été utilisé pour développer le tableau de bord de l'hélicoptère électrique Volta [1].

Les logiciels critiques avioniques doivent se conformer à la norme DO-178C, dont le supplément DO-333 encourage à utiliser les méthodes formelles pour la vérification. En particulier, il est ainsi intéressant de certifier le compilateur utilisé pour créer le logiciel. Cela consiste à démontrer formellement que la compilation n'introduit pas d'erreur dans le code généré, i. e., s'assurer de la préservation de la sémantique du programme source dans le code généré. Aujourd'hui, cette vérification est le plus souvent manuelle [4].

Nous visons à vérifier un compilateur pour SMALA, grâce à l'assistant de preuve Coq. Dans la littérature, divers travaux s'intéressent à la compilation certifiée. Ainsi, Compcert [4] est un compilateur opérationnel du langage C vérifié avec Coq ; CakeML [6] est

un projet de vérification d'un sous ensemble de StandardML avec l'assistant de preuve HOL4 ; Vélus [3] est un compilateur vérifié pour le langage réactif synchrone Lustre avec Coq. CakeML et Vélus sont en cours de développement mais déjà partiellement opérationnels. Ces trois travaux traitent de langages ayant des paradigmes de programmation différents les uns des autres. Le paradigme réactif synchrone de Lustre est celui qui se rapproche le plus du paradigme réactif de SMALA. Néanmoins, le paradigme synchrone, bien que conçu pour les systèmes réactifs, est limitant car il impose de fortes contraintes sur le temps de réaction (par exemple, la synchronisation des flux d'entrée du programme qui doivent tous donner leurs valeurs en même temps) et sur les ressources utilisées pour le développement du programme (la mémoire utilisée est bornée) [3]. SMALA est plus souple, n'imposant aucune contrainte sur le temps de réaction ni sur les ressources utilisées et apporte des aspects interactifs absents de Lustre. C'est un langage spécifique, défini via de nombreuses bibliothèques C++ dont une large bibliothèque graphique. Pour faciliter la certification du compilateur, nous visons à remplacer la plupart de ces bibliothèques C++ par des bibliothèques développées directement en SMALA. Nous avons commencé par définir un sous-ensemble très réduit que nous compléterons petit à petit, selon le besoin. Pour cela nous nous sommes affranchis de la partie interactive de Smala. Dans un premier temps, nous nous intéressons seulement au côté réactif du langage.

La section 2 présente SMALA et le sous-ensemble considéré dans cet article, qui permet d'en étudier les aspects réactifs. La section 3 développe une première formalisation de la sémantique de ce sous-ensemble. Enfin, la section 4 présente des pistes pour la suite.

2 SMALA

Le concept fondamental au cœur de SMALA est celui de processus. Un processus est une entité qui a sa propre sémantique d'exécution et un état d'activation qui peut être "activé" (le processus peut s'exécuter) ou "désactivé" (le processus attend d'être activé). Parmi ces processus, nous en présentons ici quatre qui sont essentiels et qui constituent notre sous-ensemble minimal. Le couplage (ou *binding*, noté $\rightarrow$) associe un processus source à un processus cible. Lorsqu'un processus source change d'état, il active les processus cibles associés. La *property* encapsule une valeur. L'*assignment* $=:$ permet de copier la valeur d'une *property* dans une autre. Enfin, le *connector* $\Rightarrow$ permet, à chaque activation de la *property* source, de copier sa valeur dans la *property* cible et de propager son activation. Afin d'avoir une meilleure compréhension du langage, nous en détaillons un exemple qui couvre tout le sous-ensemble présenté ici.

```
_main_
Frame f ("titre", 0, 0, 500, 400) // Création d'une fenêtre
Rectangle r (0, 0, 50, 99, 0, 0) // Création d'un rectangle
Int height (500) // Définition d'une Int Property
Exit quit (0, 1) // Processus pour quitter l'application
// L'activation de l'attribut close de f est propagée à
   quit, i.e.  fermer la fenêtre quitte l'application
f.close -> quit
// Copie la valeur de height dans r.height seulement à
   l'initialisation de l'application
height =: r.height
// La modification de la largeur de la fenêtre provoque la
   copie de cette valeur dans la largeur du rectangle
f.width => r.width
```

Un programme SMALA a une phase d'initialisation durant laquelle les processus sont déclarés et instanciés et une phase d'exécution pendant laquelle il attend un signal, provoqué par un évènement, pour déclencher une action (ci-dessus, la fermeture de la fenêtre).

3 Sémantique

La sémantique opérationnelle de SMALA reflète les phases d'initialisation et d'exécution précédemment évoquées.

3.1 Initialisation

La phase d'initialisation construit l'environnement E et une fonction S à partir du programme Ins. E associe à chaque *property* sa valeur. $E \vdash e \Downarrow v$ est l'évaluation d'une expression e en une valeur v dans E. $E, S \vdash i \rightsquigarrow E', S'$ dénote un changement d'état qui consiste en la modification de E ou S. Les règles d'initialisation sont les suivantes :

$$\text{Init}\ \frac{E, S \vdash i \rightsquigarrow E', S' \quad E', S' \vdash Ins \rightsquigarrow E'', S''}{E, S \vdash i :: Ins \rightsquigarrow E'', S''} \qquad \text{InitConnector}\ \frac{}{E, S \vdash e \Rightarrow x \rightsquigarrow E, S'} \ \text{avec } S' = \{(y, e \Rightarrow x) \mid y \in FV(e)\} \cup S$$

$$\text{EndInit}\ \frac{}{E, S \vdash [] \rightsquigarrow E, S} \qquad \text{InitProperty}\ \frac{E \vdash e \Downarrow v}{E, S \vdash Int\ x\ (e) \rightsquigarrow E[x/v], S}$$

La règle Init parcourt Ins et applique à chaque instruction la bonne règle d'initialisation.

Nous détaillons seulement les règles d'initialisation pour certains composants. Les autres règles se déduisent aisément de celles données. $E[x/v]$ associe à x la valeur v.

- La règle InitProperty exprime qu'une *property* s'évalue à l'initialisation et qu'elle n'est plus utilisée par la suite, i. e., l'instruction n'est pas ajoutée à S. $Int\ x\ (e)$ dénote la déclaration d'une *property* x avec pour valeur l'expression e.
- La règle InitConnector met à jour la fonction S en ajoutant $e \Rightarrow x$ à l'image des variables libres de e. Ainsi, l'activation d'une variable libre de e active le *connector*.

3.2 Exécution

La phase d'exécution regroupe deux ensembles de règles aux fonctionnalités différentes.

3.2.1 Règles d'exécution

Les règles d'exécution ci-dessous donnent la sémantique des processus du langage. Nous commentons ici uniquement la règle ExecConnector : l'expression e s'évalue en v dans E puis la valeur de x dans E est mise à jour en v.

$$\text{ExecBinding}\ \frac{E \vdash x_2 \rightsquigarrow E'}{E \vdash x_1 \rightarrow x_2 \rightsquigarrow E'} \qquad \text{ExecConnector}\ \frac{E \vdash e \Downarrow v}{E \vdash e \Rightarrow x \rightsquigarrow E[x/v]}$$

$$\text{ExecAssignment}\ \frac{E \vdash e \Downarrow v}{E \vdash e =: x \rightsquigarrow E[x/v]} \qquad \text{ExecProperty}\ \frac{}{E \vdash x \rightsquigarrow E}$$

3.2.2 Règles de propagation

Les règles de propagation ci-dessous définissent l'exécution d'un programme à partir du moment où celui-ci reçoit un signal.

- La règle PropagSig attend qu'un signal issu d'un évènement externe ou interne mette à jour une variable correspondant à un processus dans E. Elle exécute alors toutes les instructions déclenchées par le processus en question. La fonction S renvoie un ensemble d'instructions $S(s)$ qui constitue la file d'ensembles d'instructions I. Elle sert à générer un ordre d'exécution (les ensembles d'instructions sont ordonnés entre eux, mais les instructions au sein d'un même ensemble ne le sont pas)
- La règle PropagConnector exprime que la file d'ensembles d'instructions I complétée par l'ensemble des instructions activables par x est évaluée dans l'environnement résultant de la règle ExecConnector

$$\text{PropagSig}\ \frac{\exists s \in E,\ E[s/v], S \vdash [S(s)] \rightsquigarrow E', S}{E, S \vdash [] \rightsquigarrow E', S}$$

$$\text{PropagConnector}\ \frac{E \vdash e \Rightarrow x \rightsquigarrow E' \qquad E', S \vdash (ins :: I)@[S(x)] \rightsquigarrow E'', S}{E, S \vdash (\{e \Rightarrow x\} \cup ins) :: I \rightsquigarrow E'', S}$$

4 Suite des travaux

Cet article présente une première sémantique d'un sous-ensemble de SMALA, un langage réactif dédié à la conception de systèmes interactifs. Ce sous-ensemble n'inclut pas pour l'instant d'éléments liés à la partie interactive. Il amorce de nombreuses pistes que nous explorerons dans nos travaux futurs. Nous ajouterons au sous-ensemble les expressions, en particulier les opérations binaires, pour vérifier que notre sémantique ne génère pas d'incohérences lors de la propagation des changements (glitch) [2]. Nous formaliserons l'activation et la désactivation des processus. Enfin, nous formaliserons cette sémantique dans Coq et prouverons l'équivalence sémantique de compositions de composants.
Les différents travaux cités nous permettront tout d'abord de tester la robustesse de la sémantique et enrichir notre sous-ensemble de Smala. Par la suite, nous compilerons ce sous-ensemble en un langage impératif à définir et prouverons la correction de la compilation en s'inspirant des méthodes utilisées dans les projets CompCert, Vélus et CakeML.

References

[1] P. Antoine and S. Conversy. "Volta: the first all-electric conventional helicopter". In: *Proceedings of the More Electrical Aircraft Conference (MEA '17)* (2017).

[2] E. Bainomugisha et al. "A Survey on Reactive Programming". In: *ACM Computing Surveys* (2013).

[3] T. Bourke et al. "A Formally Verified Compiler for Lustre". In: *PLDI* (2017).

[4] X. Leroy. "Formal Certification of a Compiler Back-end or: Programming a Compiler with a Proof Assitant". In: *POPL* (2006).

[5] M. Magnaudet et al. "Djnn/Smala: A Conceptual Framework and a Language for Interaction-Oriented Programming". In: *ACM Hum-Comput. Interact* (2018).

[6] Y K. Tan et al. "A New Verified Compiler Backend for CakeMl". In: *ICFP* (2016).