ESIR
Université de Rennes 1
Campus de Beaulieu
35 042 RENNES Cedex

IRISA RENNES
Campus de Beaulieu
263 avenue du Général Leclerc
35 042 RENNES Cedex

RAPPORT DE STAGE
ÉCOLE SUPÉRIEURE D'INGÉNIEURS DE RENNES

Développement d'un générateur de code pour le développement basé sur le paradigme du model@runtime

Responsables :
Johann BOURCIER (IRISA)
Fabrice LAMARCHE (ESIR)

Auteur :
Alexandre RIO

du 01 juin au 25 septembre 2015

Remerciements

Je tiens à remercier l'IRISA et l'équipe DiverSE pour leur accueil ; en particulier Johann Bourcier, Francisco Javier Acosta Padilla et Inti Gonzales Herrera.
Je tiens également à remercier Sylvain Benhamiche, Florent Guiotte ainsi que Maxime Rivolet pour leur aide technique.

Table des matières

Introduction

0.1 L'IRISA

L'*IRISA*[1], Institut de recherche en informatique et systèmes aléatoires, est un laboratoire de recherche créée en 1975. Formé de 41 équipes réparties sur 4 pôles en Bretagne il se focalise sur «la recherche en recherche en informatique, automatique, traitement du signal et des images».

0.2 L'équipe DiverSE

L'équipe DiverSE[2], anciennement *Triskell*, est dirigée par Benoit Baudry. Elle est composée de près de 40 personnes, dont 7 permanentes, et se focalise sur la diversité dans le génie logiciel. Cette ligne directrice se traduit par des travaux de recherche dans les langages, la variabilité, l'adaptation et la diversification des logiciels.

0.3 Le stage

Ce stage est réalisé en vue de la validation de ma deuxième année à l'ESIR. À la frontière entre le monde des systèmes embarqués et celui du développement dirigé par les modèles l'objectif principal du stage est développer des outils plus génériques que ceux déjà réalisés dans le cadre de thèses réalisées au sein de l'équipe DiverSE.

Le choix d'un centre de recherche est motivé par une future année en double diplôme ESIR3 / Master Recherche en Informatique ainsi qu'une éventuelle poursuite en thèse.

1 L'Internet des Objets

L'Internet des objets, abrégé en IoT, ou Cyber Physical Systems est un réseau non pas composé d'ordinateurs ou de serveurs mais de nombreux composants bien plus petits par leurs tailles et limités en ressources.

Ce nouveau type de réseau apporte de nouvelles difficultés pour le développement de logiciel en particulier.

Ces difficultés vont de la limitation de la puissance de calcul, du stockage limité, de la durée de vie de la batterie et jusqu'à la complexité de gestion d'un réseau distribué.

Un serveur standard actuel possède un processeur au minimum quad-core cadencé à 3.5GHz, 16Go de RAM et plusieurs To de stockage disque, les nœuds principalement utilisés pour les expériences, nommés M3, sont sur une architecture ARM, leur processeur est un mono-core cadencé à 72MHz, ils possèdent 64kB de RAM, soit et 16MB de ROM. Une vue détaillée d'un nœud est en Figure 1. Outre la différence d'archicture, 64bits pour serveur et 32bits pour un nœud M3 le rapport entre les deux tailles d'espace RAM est de l'ordre de 10^6.

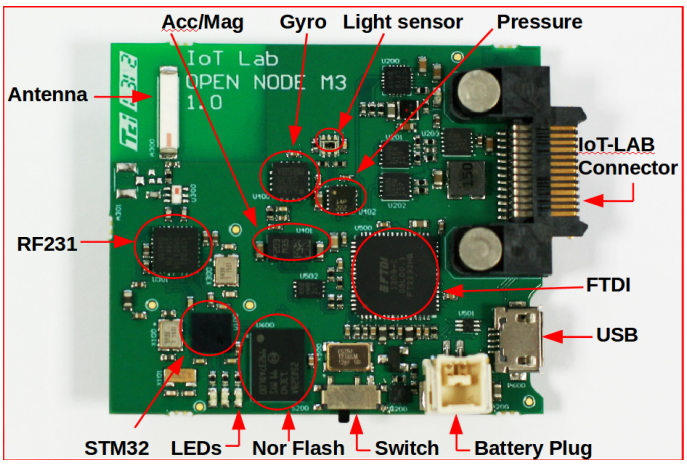


FIGURE 1 – Vue détaillée d'un nœud M3

1.1 Principales différences

De par ces différences le développement logiciel est radicalement différent de ce que l'on peut trouver lorsque l'on cible un ordinateur «classique».

Dans notre cas ces limites sont dues au langage de développement, le langage C, qui n'est pas compilé pour utiliser la *libc* standard, non disponible sur l'environnement *Contiki*. Cette restriction, entre autre, empêche l'utilisation de générateur de code standard.

Bien que possible sur un environnement *Contiki* l'allocation dynamique de mémoire ne doit pas se faire en utilisant le classique *malloc*. En effet ce dernier gère mal la fragmentation de la mémoire. Pour palier à cela *Contiki* met à disposition des développeurs d'autres méthodes telles que *memb* et *mmem*[3].

Pour rester proportionné à l'espace disque et à la puissance de calcul des nœuds *Contiki* utilise en système de fichiers spécifique nommé *Coffee* qui est une implémentation du *Contiki File System*[4]. Ce dernier empêche l'utilisation des fonctions habituelles de gestion d'accès disque telle que *fopen*.

1.2 L'environnement Contiki

Les limites physiques des nœuds composant les réseaux étudiés font qu'un système d'exploitation spécifique doit être utilisé en lieu et place du classique GNU/Linux. Le système utilisé est *Contiki*[5].

Ces nœuds sont également équipés de différents capteurs tels que :
— un capteur de luminosité,
— un thermomètre,
— un baromètre,
— un accéléromètre/magnétomètre,
— un gyromètre
Ces capteurs sont la source des données extérieures et la raison de déployer des CPS.

1.3 Réseau de nœuds

Les réseaux de capteurs ne sont généralement pas mis en réseau comme des ordinateurs classiques, où chaque nœud du réseau peut accéder directement à internet.

Ici le réseau est de type *mesh*, c'est à dire qu'un nœud est connecté à internet, il est appelé le *border router*, et que les autres sont ou reliés au *border router* ou à un autre nœud indirectement connecté au *border router*. Un exemple de ce type de réseau est montré en Figure 2.

De cette manière pour récupérer du contenu sur internet un nœud peut

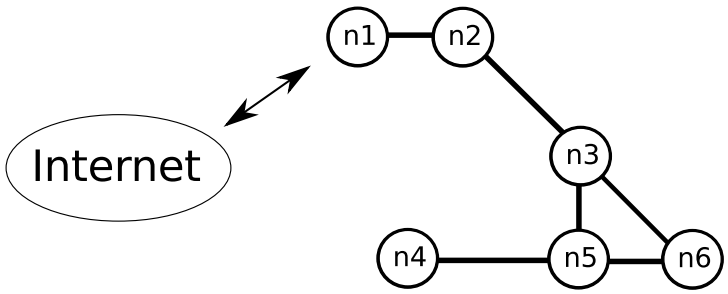


FIGURE 2 – Topologie d'un réseau mesh, ici le nœud n1 est le border routeur et est directement connecté à Internet

avoir besoin d'effectuer plusieurs bonds. Également la connexion entre nœuds n'est pas filaire mais par ondes radio, réduire au maximum les communications entre nœuds est donc un enjeu de gain de temps puisque les débits de communication sont très faible et de batterie puisque qu'une communication peut impacter tout un réseau dans certains structures.

1.4 Testbed FIT IoT-lab

FIT est un projet financé par divers organismes de l'enseignement supérieurs et de la recherche afin de mettre à disposition des entreprises et des chercheurs des structures pour les aider dans leurs travaux.

IoT-lab[6] est une infrastructure permettant l'accès à différents sites en France hébergeant du matériel dédié à l'Internet des Objets.

FIT permet l'accès à plus de 2700 nœuds répartis sur 7 différents testbed tel qu'à Rennes, Grenoble ou Rocquencourt. Les testbeds proposent jusqu'à 3 architectures différentes, dont des *m3*.

Ce grand nombre de nœuds disponibles rends possible des expériences à grande échelle dans le monde de l'IoT.

Voir annexe A.1 en page 31 pour la distribution détaillée des différents nœuds.

La réservation de nœuds se fait directement sur le site web. L'utilisateur a libre choix sur le site, le nombre et le type de nœud. Dans la limite des ressources disponibles et pour une durée conseillée d'au maximum 2 heures.

Une fois des nœuds réservés et obtenus il est possible d'y déployer un nouveau firmware. Toutes ces opérations, de la réservation de nœuds au flashage

d'un nouveau firmware, sont directement réalisable en ligne de commande [1].

2 model@runtime, Kevoree et KMF

2.1 model@runtime

Le développement de réseaux distribués de machines de plus en plus complexes a amené la volonté et le besoin de pouvoir les reconfigurer plus aisément. L'intérêt peut être de mettre à jour un module fonctionnant sur une machine tout en minimisant le temps durant lequel le module ne sera pas fonctionnel.

Pour cela le paradigme du *model@runtime* utilise un modèle servant à définir l'architecture détaillée d'un réseau à un instant *t*. Ce modèle permet également de déployer ou de reconfigurer un système pendant son exécution.

Cette approche fonctionne sur des réseaux d'ordinateurs et de serveurs [7] et l'objectif de la thèse de Francisco Javier Acosta Padilla est de prouver que cette approche fonctionne également pour l'Internet des Objets.

2.2 KMF

KMF, Kevoree Modeling Framework [2], est un langage de modeling s'inspirant de *EMF*, Eclipse Modeling Framework [3], mais conçut pour répondre aux besoins du *model@runtime*.

L'objectif de *KMF* est de, à partir d'un méta-modèle, produire une structure de données et des outils de manipulation.

2.3 Kevoree

Kevoree est un projet open-source implémentant le paradigme du *model@runtime*. Les versions principales sont écrites en *Java* et en *Javascript* mais des implémentations en *C* et *C++* ont également été implémentées. Chaque version de *Kevoree* est basée sur une version de *KMF*.

1. https ://www.iot-lab.info/tutorials/experiment-cli-client/
2. http ://kevoree.org/kmf/
3. http ://www.eclipse.org/modeling/emf/

Kevoree offre des mécanismes de comparaison et de fusion de modèles pour permettre l'adaptation de systèmes distribués.

2.3.1 Kevoree-c

La version utilisée durant ce stage est *kevoree-c*[4] réalisée par Francisco Javier Acosta Padilla dans le cadre de sa thèse. Contrairement aux autres versions de *Kevoree* celle-ci n'a pas été obtenue par génération depuis un méta-modèle mais a été écrite à la main.

4. https ://github.com/kYc0o/kevoree-c-reloaded

3 Travaux préliminaires

3.1 Mise en place de l'environnement de développement

Pour mieux appréhender la variété de l'environnement et des outils la première tâche qui m'a été confiée n'était pas au cœur de ses systèmes. J'ai tout d'abord lu l'article [8] pour me familiariser avec cet environnement.

Les nœuds *m3* étant uniquement capable d'exécuter du code déjà compilé il a fallu que je mette en place un environnement dit de *Cross-compilation*, ou compilation croisée, qui consiste à produire un code binaire exécutable par une machine ayant une architecture différente de celle servant au développement.

Pour développer un nouveau module pour *Contiki* il suffit d'en récupérer les sources [5], de se placer dans le répertoire contenant les sources d'un module et de le compiler en précisant l'architecture cible, iotlab-m3.

```
git clone https://github.com/iot-lab/contiki
cd contiki/example/hello-world
make TARGET=iotlab-m3
```

Listing 1 – Résumé des étapes de compilation

Cette compilation produira un firmware contenant *Contiki*, le module courant ainsi que tout autre module marqué en dépendance. Ce firmware pourra directement être flashé sur un nœud *m3*.

3.2 Compression des modèles

Pour pouvoir s'adapter et selon le principe du *model@runtime* chaque nœud d'un réseau IoT doit comparer le nouveau modèle du réseau avec le courant et en déduire s'il doit se modifier. Il est donc nécessaire que le nouveau modèle soit échangé entre tous les nœuds. Dans un soucis de temps de développement et puisque cela n'impacte pas l'idée de *model@runtime* l'implémentation de *kevoree-c* ne compresse pas les modèles avant de les transmettre aux pairs voisins.

La compression permettrait aux nœuds de manipuler des modèles représentant plus d'instance et plus de composant dans un même fichier et donc être capable de gérer des réseaux plus grands.

5. https ://github.com/iot-lab/contiki

3.3 État de l'art

Puisque aucun algorithme de compression de données n'est fourni avec *Contiki* j'ai d'abord commencé par en chercher implémentés en *C*, recommandés pour les systèmes embarqués et disponibles en sources libres dans des licences permettant leurs réutilisations.

L'objectif étant d'avoir le meilleur taux de compression tout en restant le moins demandeur de ressources matérielles, en particulier en mémoire RAM pour pouvoir être exécuté sur les nœuds *M3*.

Les implémentations retenues et étudiées sont *Huffman*, *fastlz*[9] et *lzf*[10].

Les résultats ont été obtenus en utilisant l'outil *Massif*[11] de la suite d'outils Valgrind. *Massif* permet de mesurer l'utilisation de différentes zones mémoire lors de l'exécution d'un programme. Par défaut seul l'utilisation du tas est mesuré mais la pile peut également l'être après configuration. Les fichiers de données produits sont binaires mais peuvent être visualisés à l'aide de l'outil *msprint* fourni avec *Massif*.

Les mesures étant faites sur un ordinateur classique d'architecture 64bit et non sur l'environnement de production la compilation a été réalisée en 32bit pour réduire la taille des adresses de pointeur manipulées et ainsi avoir des résultats plus fiables. Tous les algorithmes ont été utilisés en compression et en décompression sur plusieurs modèles différents représentatifs de ceux utilisés lors des expériences.

Les scripts utilisés pour automatiser les mesures et produire les résultats suivants sont disponibles sur Github [6].

Les utilisations de mémoire étant bien trop important, voir Image 3.5, pour notre utilisation en IoT un algorithme de compression spécifique a été développé. De plus réduire manuellement l'utilisation de mémoire des programmes réduit drastiquement leur efficacité.

3.4 Algorithme spécifique

Vue qu'aucune implémentation existante ne peut fonctionner sur *Contiki* j'ai développé un programme de compression.

La structure générale et les mots-clés du fichier à compresser étant connus à l'avance la méthode la plus simple de compression est de réaliser une table d'association entre les mots les plus fréquemment utilisés et de les substituer

6. https ://github.com/AlexandreRio/embedded-compression-benchmark

par un code plus court.

Cette approche oblige tous les nœuds à connaître la table d'association utilisée pour compresser lors de la décompression. Cette table étant constante lors de la vie du programme elle peut être stockée en ROM, ce qui est important dans notre cas.

Pour obtenir cette table d'association j'ai réalisé un programme *Java*, disponible sur Github [7], qui se base sur un ensemble de modèle. Le programme peut être paramétré pour produire une table plus au moins grosse, au détriment du taux de compression moyen.

Les premiers tests sur *Contiki* ont d'abord consisté à simplement compressé une chaine de caractères constante au programme, puis un fichier local au nœud et pour finir un fichier local au nœud envoyé à un second nœud qui le décompresse après réception. Le résultat attendu étant que le fichier obtenu après compression, transmission et décompression soit identique ou au moins sémantiquement identique.

Les expériences ont été réalisées sur le testbed FIT IoT-lab de Rocquencourt qui a l'avantage de ne compter que 24 nœuds M3 (voir annexe A.1 en page 31), ce qui est largement suffisant dans le cadre de cette expérience et permet d'être sur d'avoir des nœuds disponibles et de pouvoir les réserver pour plusieurs heures sans gêner d'autres expériences.

La transmission de données, ici de modèles, est particulière sur un réseau mesh et dans le cas de *Kevoree*. Pour le *model@runtime* la transmission de données est utilisée pour transmettre le nouveau modèle depuis 1 nœud à $n-1$ nœuds, cette transmission a été appelée dissémination. Dans le cadre de mes expériences de test de compression avant transmission j'ai déployer plusieurs nœuds *border router* possédant la nouvelle version du modèle et seulement 1 possédant l'ancienne version. De cette manière et en prenant en compte comment l'algorithme de dissémination est implémentée le nœud possédant l'ancienne version pourra recevoir la nouvelle, découpées en morceaux, de tous les nœuds proches de lui et ainsi le recevoir plus vite et donc réduire le temps des expériences et par extension mon temps de développement.

Après avoir résolu les différents bugs liés à l'environnement, tel que le système de fichier, le code du programme de compression a déplacer afin de produire un module indépendant dans *Contiki*. De ce fait n'importe quel autre module ou programme basé sur *Contiki* peut utiliser cette fonction pour compresser ou décompresser des fichiers.

7. https ://github.com/AlexandreRio/Contiki-compression-tools

Les sources de la version de compression seule [8] et de la compression et transmission [9] sont disponibles sur Github.

3.5 Résultats mesurés

Le taux de compression $\frac{taille\ finale}{taille\ initiale}$ moyens de chaque algorithme est mesuré sur un ensemble de fichier 14 fichiers représentant 7 modèles différents. Chaque modèle est représenté par un fichier facilement lisible par l'Homme et par une représentation minimaliste, sans caractère de mise en forme superflu.

Algorithme	*Taux de compression*
LZF	0.174
$Huffman$	0.604
$Fastlz - 1$	0.179
$Fastlz - 2$	0.171
Algorithme spécifique	0.641

L'annexe A.3 en page 33 détaille les tailles avant et après compression pour l'ensemble de ces fichiers mais le taux de compression moyen est de 0.641 tout en ayant une utilisation de mémoire minime. La Figure 3.5 montre l'utilisation moyenne de 3Ko de RAM en compression et en décompression.

8. https ://github.com/AlexandreRio/contiki/tree/usingOldKevoree/examples/compress
9. https ://github.com/AlexandreRio/contiki/tree/usingOldKevoree/examples/compress-disseminate

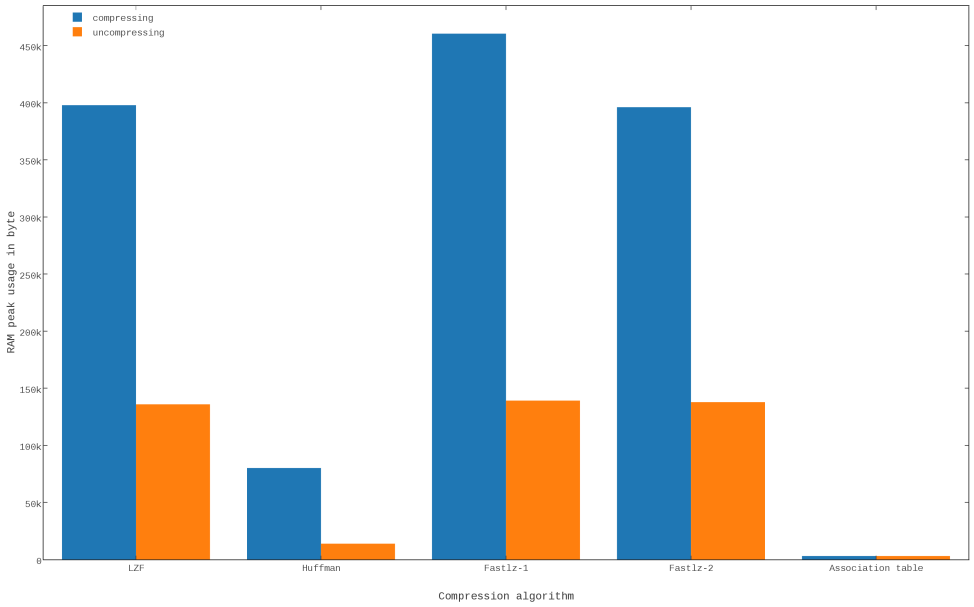


FIGURE 3 – Pic moyen d'utilisation de la RAM en octet en fonction de l'algorithme de compression.

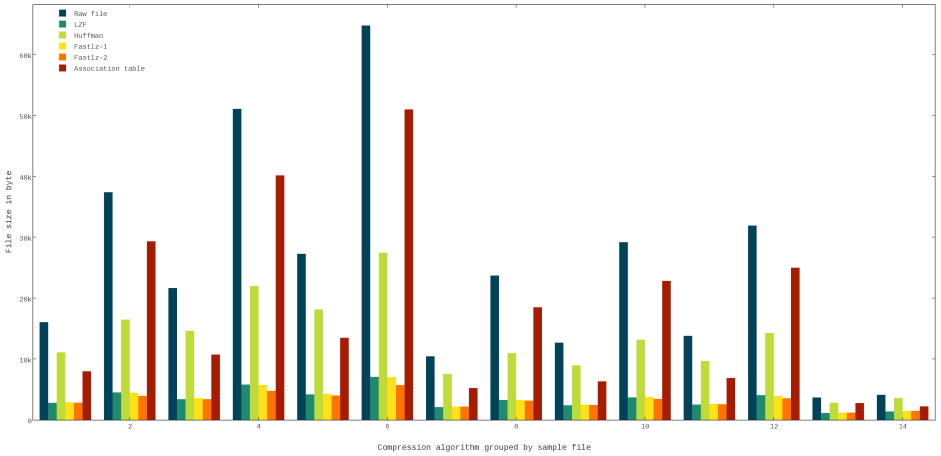


FIGURE 4 – Mesure de compression de modèles à l'aide de différents algorithmes

3.6 Parser spécifique

Un des points crucial de *Kevoree-c* est la possibilité de reconstruire un état mémoire depuis un fichier représentant un modèle. Cette phase de déserialisation repose sur une partie critique du programme. En effet la version actuelle utilisée pour les expériences, en plus de ne fonctionner qu'avec une version fixe de *KMF*, a été réalisée dans des contraintes de temps forte et ne reconnaît qu'une partie minimale du méta-modèle. Le code produit ne suivant pas le patron de conception et pour les raisons énoncées le code source est difficilement maintenable.

Pour palier à ces défauts et mieux comprendre les mécanismes internes de *Kevoree-c* j'ai donc commencé à étudier les possibilités de ré-écriture du dé-serializer.

Il devait répondre aux contraintes suivantes :

— avoir une faible empreinte mémoire RAM,
— avoir le moins de traitement ad-hoc à la structure de données,
— être facile à relire et modifier

Mon premier réflexe a été de vouloir utiliser un générateur de parser, de cette manière seule la grammaire aurait été à écrire, le parser en *C* aurait été généré par l'outil.

3.6.1 AntlrV3

AntlrV4 est le générateur de parser utilisé pendant le module de compilation d'ESIR2, dans sa version 4 il ne permet pas/plus de générer du *C* c'est pourquoi j'ai testé AntlrV3.

Pour vite avoir une idée du coût mémoire d'Antlr j'ai utilisé une simple grammaire reconnaissant le format JSON [10] en partant d'un exemple de projet *C* pour AntlrV3 [11].

Malheureusement même avec une grammaire reconnaissant uniquement un fichier *JSON* et n'effectuant aucune action le coût mémoire est trop important. Sans compter le poids du parser généré l'ensemble des fichiers à inclure pour le faire fonctionner pèse déjà plus de $400Kb$.

10. https ://github.com/antlr/grammars-v4/blob/master/json/JSON.g4
11. https ://github.com/antlr/examples-v3/tree/master/C

3.6.2 Flex et Bison

Les outils de génération de parser les plus connus sont : *Flex*[12], pour l'analyse lexicale et *Bison*[13] pour l'analyse syntaxique

Ensemble ils peuvent produire du code *C* de taille cette fois bien plus raisonnable. En suivant la même démarche que pour AntlrV3 j'ai d'abord créé un parser pour uniquement reconnaître la syntaxe JSON.

3.6.3 Portage et entrées/sorties

Le parser lit son fichier d'entrée à l'aide des méthodes *fopen* et *fclose* manipulant des types *C* : *FILE*. Le problème étant que *Contiki* n'utilise pas ce standard d'entrées/sorties mais des interfaces spécialisées pour son système de fichiers.

Flex et *Bison* ne proposent aucune option pour changer la façon dont le parser va lire le fichier d'entrée, de plus le code généré est tellement optimisé que le modifier directement est très difficile et long.

3.6.4 Portage et tableaux

Réutiliser des parties du code généré c'est également avéré être une tâche impossible, pour les mêmes raisons que précédemment. En écrivant une lecture du disque propre à *Contiki* et en utilisant le code du traitement produit par *Flex* et *Bison* le programme résultant peut être compilé mais produit une erreur au moment de l'édition des liens due à une incompatibilité dans la *libc* de *Contiki*

Le traitement de ce genre d'erreur étant trop chronophage le développement d'une solution sur mesure est plus judicieux.

3.6.5 Solution sur mesure

J'ai donc commencé à écrire un lexer et un parser prenant en entrée un fichier JSON et créant les instances *C* correspondant.

En manipulant les instances en question, comment sont stockés les attributs, où se situe le code des méthodes, comment se passe l'appel des méthodes et à force d'utiliser *Contiki* j'ai pu commencer à être à l'aise avec cet environnement, ce qui était l'objectif de cette tâche et non la réalisation d'un parser spécifique.

Les nombreux problèmes de mémoire rencontrés dus à l'allocation dynamique de *Contiki* ont fait que la tâche à été abandonnée puisqu'un parser générique doit être écrit dans la suite du stage.

4 kmfc : un générateur de code

Après m'être familiarisé avec le système *Contiki*, le paradigme du *model@runtime* et son implémentation *Kevoree-C* j'ai pu commencer l'objectif principal de mon stage, à savoir produire dynamiquement les parties critiques de *Kevoree-C* à partir d'un méta-modèle *Kevoree*.

4.1 Intérêt de générer du code

Comme décrit en partie 2.3 l'actuelle version de *Kevoree-C* a été écrite manuellement. En plus d'avoir une série d'inconvénients, l'écriture manuelle a également quelques avantages que la ré-écriture générique se devra d'essayer de conserver.

Ces inconvénients sont en partie les mêmes que ceux détaillés dans la partie 3.6 et son avantage principal découle de ces défauts, en étant si spécifique à la version du méta-modèle lu et à la structure de données, le programme final est très performant. L'objectif va être de produire du code le plus optimisé possible même s'il sera théoriquement difficile d'atteindre ce même niveau. Ici la notion d'optimisation est subjective, elle consiste plus à évaluer la manière dont le stockage des données est réalisés, les éventuelles redondances d'information et la complexité algorithmique des sections critiques.

Il est par exemple évident que le code source du générateur de code doit privilégier la lisibilité à la rapidité d'exécution.

4.2 Méta-modèle *Kevoree*

Le méta-modèle de *Kevoree* peut être représenté de la même manière qu'un méta-modèle *EMF*, *Eclipse*[14] et en particulier sa version *Eclipse Modeling Tools*[15] peut être utilisé pour l'afficher.

L'extrait de diagramme en Figure 5, page 19 et l'Annexe A.4 montrent que les classes possèdent généralement peu d'attributs primitifs. Ils permettent également de remarquer que de nombreuses classes implémentent l'interface *NamedElement* et obtiennent donc un attribut *name* et que toutes les classes sont référencées directement ou indirectement par la classe *ContainerRoot*.

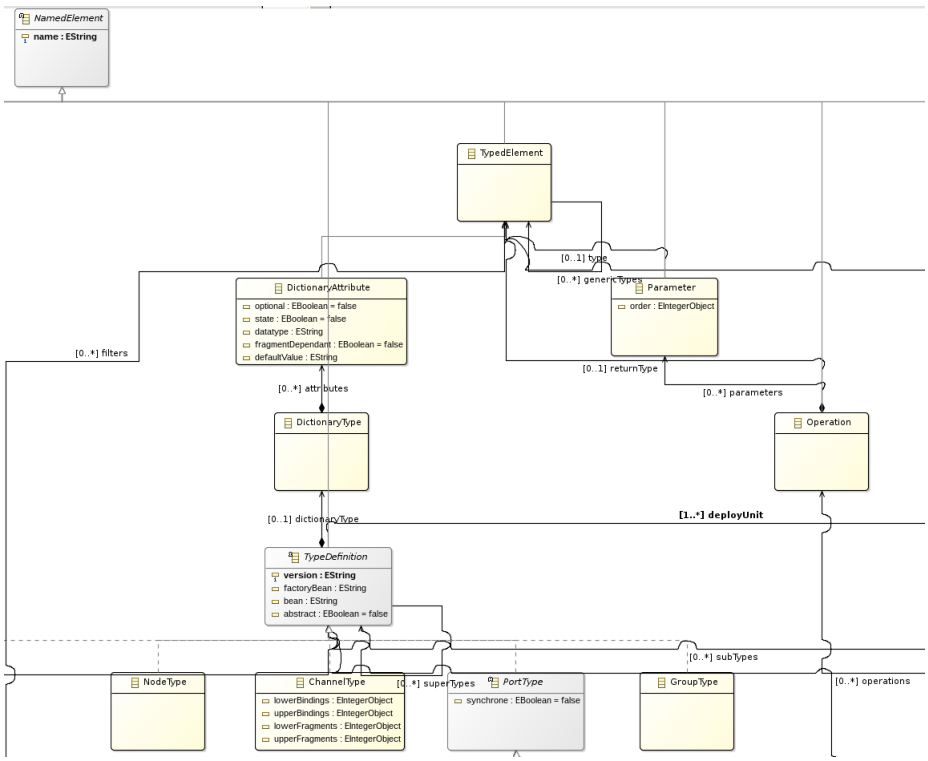


FIGURE 5 – Extrait du méta-modèle de Kevoree v4 écrit en KMF2

4.3 Structure de *Kevoree*

Pour résumer, *Kevoree-c* est composé d'une structure de données détaillée en section 4.2, d'outils de manipulation pour permettre la sérialisation/désérialisation, comparaison de modèles et permettre l'adaptation suite au calcul de ses différences.

4.4 Flux de travail

La figure 6 en page 20 résume les différentes étapes jusqu'à la création du nouveau firmware et l'utilisation du méta-modèle *Kevoree-c* et du modèle représentant les instances et leurs modules. Le compilateur utilisé pour cibler une machine personnelle est *gcc* et pour cibler les nœuds *m3* sa version de compilation croisée *msp430-gcc*.

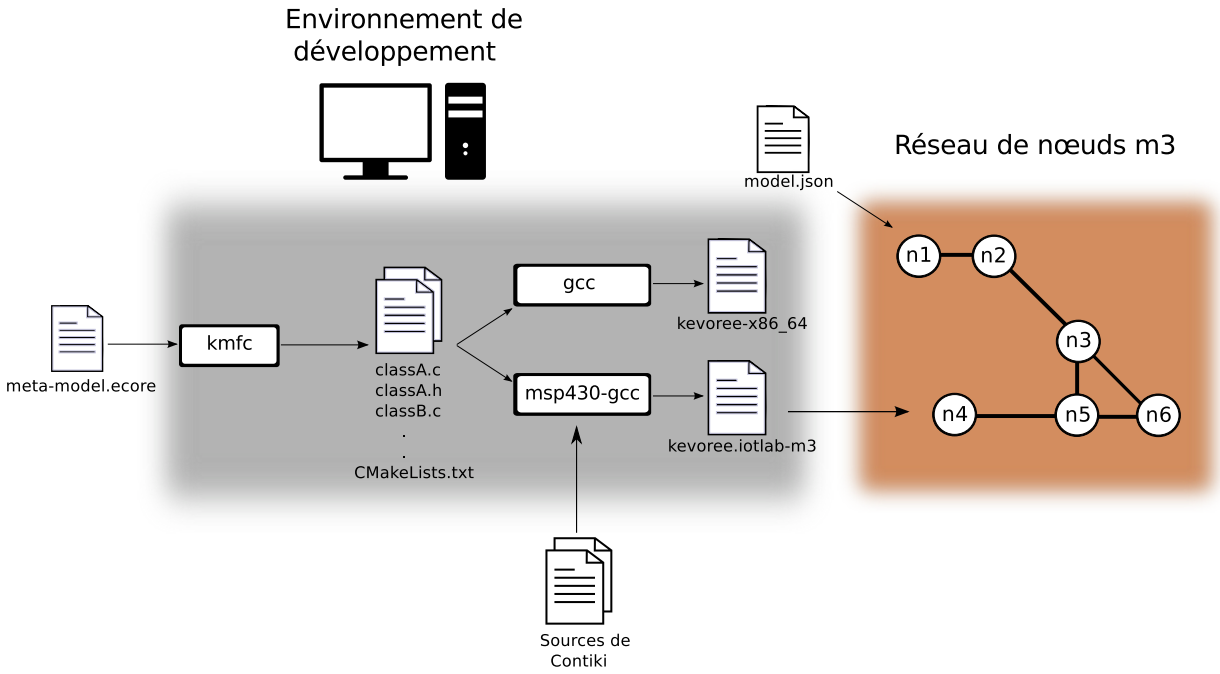


FIGURE 6 – Flux de travail du développement au déploiement du nouveau firmware.

4.5 kmfcpp

Avant de commencer mon générateur de code Johann BOURCIERet Francisco Javier ACOSTA PADILLA m'ont indiqué de regarder la version *C++* [12] de *KMF*. Bien que le langage de destination ne soit pas le même le format du méta-modèle d'entrée est identique, c'est une partie qui peut donc être réutilisée.

Le langage *C++* permet nativement beaucoup de mécanismes utiles à la représentation d'une telle structure de données, à savoir :

— le paradigme d'orienté objet qui permet une encapsulation des attributs et des méthodes,
— l'héritage et le polymorphisme, qui sont utilisés dans le méta-modèle,
— une simplification de la gestion de la mémoire allouée dynamiquement, avec les destructeurs de classes

Ces fonctionnalités natives font que *kmfcpp* ne génère que peu de code et peut le faire en ne parcourant le méta-modèle qu'une seule fois et en produisant directement le code *C++* équivalent.

12. https ://github.com/kevoree/kmfcpp

Pour générer du code *kmfc* utilise le framework *Velocity*[16], il permet de créer des templates de texte contenant des variables ou des structures de contrôle qui seront évaluées et remplacées lors du parsing du méta-modèle.

L'utilisation d'un moteur de template permet de séparer la logique algorithmique du générateur du code même généré. La lisibilité en est grandement améliorée.

4.5.1 L'utilisation de template

Comme visible sur le Listing 2 en page 21 la syntaxe de *Velocity* permet de gagner en lisibilité. Les instructions de pré-processeur *C*, par exemple lignes 1, 3 et 4, doivent être explicitement ignorées en étant encadrées par `#[[ ]]#`, `#include` étant également une instruction *Velocity*.

Les variables *Java* sont référencées en écrivant `$variable`, par exemple ligne 14. Une variable de type primitif sera directement affiché, les objets se verront automatiquement appelé leur méthode `toString()`.

Velocity permet également d'appeler les méthodes des objets, comme en ligne 9.

Pour finir, les structures de contrôle permettent de calculer uniquement une liste du côté *Java* et de la passer à *Velocity* qui se contentera de la parcourir, comme visible en ligne 24 à 26.

```
1  #[[#include "]]#$name#[[Test.h"]]#
2
3  #[[#include <stdlib.h>]]#
4  #[[#include <check.h>]]#
5
6  #foreach ($test in $testSuite.keySet())
7  START_TEST ($test)
8  {
9  $testSuite.get($test)
10 }
11 END_TEST
12
13 #end
14 Suite* $name#[[_suite(void)]]#
15 {
16     Suite *s;
17     TCase *tc_core;
18
19     s = suite_create("$name");
20
21     /* Core test case */
22     tc_core = tcase_create("Core");
23
24 #foreach( $test in $testSuite.keySet() )
25     tcase_add_test(tc_core, $test);
```

```
26 #end
27     suite_add_tcase(s, tc_core);
28
29     return s;
30 }
```

Listing 2 – Fichier testSource.vm permettant de générer un fichier de test unitaire.

4.6 Un vrai compilateur

Les premières versions de *kmf* adoptaient plus une logique de transcription de méta-modèles vers le langage *C*, à l'image de *kmfcpp* la génération de code était réalisée directement pendant le parcours du méta-modèle d'entrée.

Un compilateur se différencie par sa représentation intermédiaire. Le travail du compilateur est alors séparé en deux parties, une face avant et une face arrière. La face avant est chargée de lire le langage d'entrée, ici un méta-modèle et d'en construire une représentation intermédiaire propre au compilateur. La face arrière se charge de générer du code, ici du *C* à partir de cette représentation.

La représentation intermédiaire permet de séparer les préoccupations et offre de nombreux avantages.

Parmi ces avantages, les suivants peuvent nous concerner :
- il est plus aisé de corriger ou comment le modèle est lu ou comment le code *C* est généré,
- l'optimisation s'effectuera sur la représentation intermédiaire et ne sera pas dépendante des formats lus et produits,
- lire un nouveau format de méta-modèle n'impacte qu'au maximum la moitié du code,
- de même que cibler un nouveau langage ou une variante de *C*

En plus d'un enseignement de Compilation suivi à l'ESIRje me suis documenté[17], [18] durant la rédaction de ce compilateur.

4.7 Générer une structure de données

La première étape est donc, petit à petit, comprendre la manière dont *Kevoree-c* a été écrit pour pouvoir «lier» les parties de code au méta-modèle. De manière, générale pour implémenter les paradigmes nécessaires tels que détaillés dans la section 4.5 en optimisant la taille du binaire produit l'utilisation de pointeurs sur variables et sur fonctions est primordiale.

4.7.1 Construction d'une classe

Une classe du méta-modèle est représenté par un fichier `.h` permettant d'exposer des informations au reste du compilateur et un fichier `.c` contenant entre autre l'implémentation des méthodes métiers.

Une classe *C* est donc composée de :
— une `struct` pour stocker tous les attributs,
— une autre `struct` représentant une *Table Virtuelle*. Elle contient toutes les fonctions que la classe est censée posséder sous forme de pointeurs sur fonctions.

La `struct` des attributs possède un pointeur sur la `struct` représentant la table virtuelle, de cette manière elle peut représenter la classe et pour une question de clarté, porte son nom.

La contrainte est de connaître la taille de chaque attribut au moment de la compilation. Les types primitifs ne posent pas de problèmes pour avoir un équivalent en *C*, les relations unaires sont représentés par un pointeur sur la `struct` des attributs de la classe voulue. En revanche les relations multiples du méta-modèle du type `0-*` produirait un tableau de taille variable, la solution est de gérer cette mémoire de façon dynamique. Cette préoccupation a déjà été résolue dans *Kevoree-C* en développant la structure de données `map_t`.

4.7.2 Gestion de l'héritage

Grâce à ces deux `struct` l'héritage est géré de manière simple, si une classe `B` hérite d'une classe `A` alors tous les attributs de `A` seront copiés dans la `struct` de `B`.

Pour l'appel de fonctions définies dans une classe mère plusieurs approches sont possibles.

La première est d'hériter des pointeurs de fonctions parents en les copiant et en les faisant pointer sur le code contenu dans les classes mères en passant par l'attribut `super` pointant sur la classe mère. De cette manière le code de la méthode n'est pas dupliqué mais le compilateur ne peut déterminer l'emplacement de code à la compilation et ne peut pas initialiser la `struct` en la laissant constante. C'est l'approche utilisée actuellement dans *kmfc*.

Une autre approche est d'ajouter des fonctions qui vont elles se contenter d'appeler le code parent. C'est l'approche utilisée dans la version de *Kevoree-C*.

4.7.3 Gestion du polymorphisme

Le langage *C* offre une possibilité d'outre passer la vérification des types des pointeurs. En utilisant la notation `void*` et des casts il est possible d'utiliser le polymorphisme.

L'utilisation de `void*` pour désigner un espace mémoire rend difficile la relecture du code, l'avantage réside dans le fait que le code est généré et ne doit en théorie pas être relu ou modifié directement. Les modifications sont à faire dans le générateur, où le code est isolé et le rôle de chaque variable facile à déterminer.

4.7.4 Exemple d'implémentation

L'annexe A.5 en page 35 résume l'héritage des attributs, leur typage, la gestion de la table virtuelle et des déclarations des signatures de fonction.

4.7.5 Générer des tests unitaires

En plus de générer une structure de données j'ai estimer que générer du code unitaire pour cette structure avait des avantages. Pour chaque méthode de chaque classe je génère au moins un test unitaire.

En plus de vérifier que l'appel des méthodes modifient l'état de l'objet leur rôle est surtout de s'assurer que les appels peuvent être réalisés. Un appel de pointeur de fonctions mal initialisé résulterait par un échec du test.

Aucun test n'était réalisé sur *Kevoree-C*.

4.8 Générer des outils de manipulation

En plus de la structure de données il faut pouvoir effectuer les actions nécessaires au paradigme du *model@runtime*.

La sérialisation des instances courantes est possible par simple appel à une méthode de la classe *ContainerRoot*, l'appel est ensuite propagé aux objets référencés, comme implémenté dans *Kevoree-C*. Une approche utilisant le patron de conception Visiteur[19] a été implémenté mais le format JSON demande un traitement particulier pour les éléments en début et fin de liste alors que Visiteur demande à ce que chaque objet effectue son action indépendamment. Par exemple une liste comment par un `[`, finit par `]` et chaque élément, sauf le dernier, est suivi d'une virgule.

La désérialisation est pensée de manière totalement différente de celle implémentée dans *Kevoree-C* pour palier à ces défauts, voir section **??**. Quelques fonctions génériques telles que `parseObject` et `parseArray` sont appelées en fonction du contexte pour récupérer les informations voulues du fichier JSON. Le contexte, les attributs attendus et la hiérarchie à parcourir, sont définis par un ensemble de tables tel que le Listing 3.

```
1 const struct at NodeNetwork_Attr[NodeNetwork_NB_ATTR] = {
2 {"eClass", doNothing, PRIMITIVE_TYPE, PRIMITIVE_TYPE},
3 {"generated_KMF_ID", NodeNetworkSetgenerated_KMF_ID, NODENETWORK_TYPE, ↩
      PRIMITIVE_TYPE},
4 {"link", parseArray, NODENETWORK_TYPE, NODELINK_TYPE},
5 {"initBy", NodeNetworkSetinitBy, NODENETWORK_TYPE, CONTAINERNODE_TYPE},
6 {"target", NodeNetworkSettarget, NODENETWORK_TYPE, CONTAINERNODE_TYPE},
7 };
```

Listing 3 – Exemple de structure permettant de connaître le schéma de désérialisation.

L'intérêt est de n'écrire le nom de l'attribut qu'une fois. Un pointeur de fonctions lui est associé et est appelé lorsque cet attribut est rencontré ainsi que deux types énumérés pour définir un contexte, le premier identifie la classe à laquelle l'attribut appartient, le second le type de l'objet à lire s'il s'agit d'un lien multiple.

Les mécanismes de comparaison et d'adaptation ne représentent pas la partie critique de *Kevoree-C*, leurs algorithmes seront donc repris et incorporés à *kmfc*.

4.9 Générer le reste

Pour avoir un projet complet et fonctionnel d'autres fichiers doivent également être inclus ou générés. Il s'agit de fichier correspondant à des types tels que `map_t` ou pour gérer le format JSON. Ces fichiers sont simplement ajoutés au code généré.

Des fichiers lanceurs, contenant une méthode `main`, sont également copiés, il me permette de développer une fonctionnalité d'abord sur quelques classes choisies avant de l'ajouter au compilateur et de gérer l'ensemble des classes couvertes par le méta-modèle.

D'autres fichiers sont nécessaires à la compilation du projet. La compilation est assistée par *cmake*[20] pour gérer facilement la production de différents exécutables : un pour chaque fichier contenant une méthode `main`, un pour lancer la suite de tests. Cmake permet de n'inclure les tests que dans

l'exécutable lançant les tests.

Conclusion

Avancement

La réalisation d'un compilateur est une tâche longue et complexe, la génération de code implique une confiance dans le générateur, c'est pourquoi il doit être le plus lisible possible. Tant dans sa représentation intermédiaire que dans sa manière de générer du code.

À la rédaction de ce rapport, soit 3 semaines avant la fin du stage :

— le désérialiseur est partiellement implémenté, seul reste la gestion des liens `1-1` entre les classes,

— l'intégration des outils de comparaison et d'adaptation développés pour *Kevoree-C* reste à faire

Si *kmfc* génère du code *C* à partir d'un méta-modèle, certains cas sont ad-hoc au méta-modèle *Kevoree*. Ces cas ont été documentés et expliqués directement dans le code source sous forme de commentaires.

Bilan

Un compilateur fait le lien entre deux langages, dans le cadre de mon stage il s'agissait de lire des méta-modèles pour produire du code *C* destiné à des systèmes embarqués. Bien que le code produit soit lui aussi destiné à être compilé il n'en reste pas moins de très bas niveau et demande de bonnes connaissances tant du langage en lui même que des systèmes exécutant le firmware produit. Le passage d'un langage de haut niveau de modélisation à un langage très proche du matériel est une tâche très stimulante. À cela s'ajoute le fait de manipuler plusieurs langages de programmation, celui d'entrée du méta-modèle, celui dans lequel est implémenté le compilateur, *Java*, celui du code généré, *C*, sans oublier les différents outils venant se greffer au projet tels que le moteur de template *Velocity* et *cmake*.

Perspectives

La structure de *kmf* permet de facilement ajouter en retirer des méthodes à générer. Il est donc possible de rapidement tester un nouveau firmware que l'on peut déployer directement sur le testbed.

En plus de pouvoir expérimenter des modifications sur le méta-modèle *Kevoree* et obtenir immédiatement un le code *C* correspondant *kmfc* per-

met également d'expérimenter des représentations de données ou d'autres manières d'implémenter le *model@runtime* en ne modifiant qu'une partie minime du code source.

Dans l'optique d'optimisation il serait possible d'intégrer *kmfc* à un autre outil afin de mesurer l'utilisation de la RAM en fonctionnement et la taille en ROM du firmware produit.

Références

[1] Institut de recherche en informatique et systèmes aléatoires. [Online]. Available : http://www.irisa.fr/

[2] Diverse team | software diversity for modeling and testing. [Online]. Available : http://diverse.irisa.fr/

[3] Memory allocation · contiki-os/contiki wiki. [Online]. Available : https://github.com/contiki-os/contiki/wiki/Memory-allocation

[4] File systems · contiki-os/contiki wiki. [Online]. Available : https://github.com/contiki-os/contiki/wiki/File-systems

[5] Contiki : The open source os for the internet of things. [Online]. Available : http://contiki-os.org

[6] Fit/iot-lab • very large scale open wireless sensor network testbed. [Online]. Available : https://www.iot-lab.info/

[7] F. Fouquet, "Model@Runtime for continuous development of heterogeneous distributed adaptive systems," Theses, Université Rennes 1, Mar. 2013. [Online]. Available : https://tel.archives-ouvertes.fr/tel-00831018

[8] F. J. Acosta Padilla, F. Weis, and J. Bourcier, "Towards a Model@runtime Middleware for Cyber Physical Systems," in *Proceedings of the 9th Workshop on Middleware for Next Generation Internet Computing*, Bordeaux, France, Dec. 2014. [Online]. Available : https://hal.inria.fr/hal-01090269

[9] Fastlz - lightning-fast compression library. [Online]. Available : http://fastlz.org/

[10] Marc lehmann's "liblzf". [Online]. Available : http://oldhome.schmorp.de/marc/liblzf.html

[11] Valgrind. [Online]. Available : http://valgrind.org/docs/manual/ms-manual.html

[12] flex : The fast lexical analyzer. [Online]. Available : http://flex.sourceforge.net/

[13] Gnu bison. [Online]. Available : https://www.gnu.org/software/bison/

[14] Eclipse - the eclipse foundation open source community website. [Online]. Available : http://www.eclipse.org/home/index.php

[15] Eclipse modeling tools | packages. [Online]. Available : http://www.eclipse.org/downloads/packages/eclipse-modeling-tools/marsr

[16] Apache velocity site - the apache velocity project. [Online]. Available : https://velocity.apache.org/

[17] A. Aho, M. Lam, R. Sethi, and J. Ullman, *Compilateurs : principes, techniques et outils : Avec plus de 200 exercices*. Pearson, 2007. [Online]. Available : https://books.google.fr/books?id=iN9TNwAACAAJ

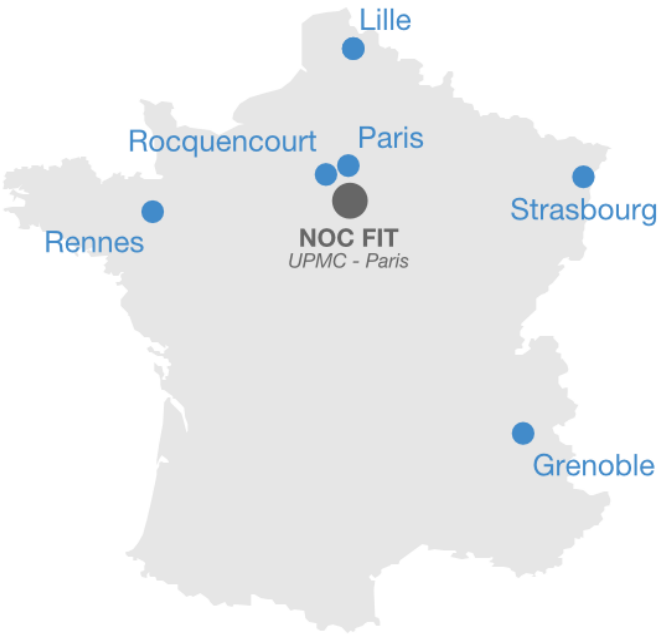
[18] A. W. Appel and J. Palsberg, *Modern Compiler Implementation in Java*, 2nd ed. New York, NY, USA : Cambridge University Press, 2003.

[19] E. Freeman, E. Freeman, K. Sierra, and B. Bates, *Head First Design Patterns*, ser. Head first series. O'Reilly Media, 2004. [Online]. Available : https://books.google.fr/books?id=LjJcCnNf92kC

[20] Cmake. [Online]. Available : http://www.cmake.org/

Annexes

A.1 Distribution des nœuds FIT iot-lab



Testbeds distribution of nodes:

	Grenoble	Lille	Rocquencourt	Strasbourg	Rennes	Paris	Total
WSN430 Node (800MhZ)	256	-	-	256	-	-	**512**
WSN430 Node (2.4GhZ)	-	256	120	-	256	-	**632**
M3 Node	384	320	24	120	-	90	**938**
A8 Node	256	-	200	24	-	70	**550**
Open Host Node	32	64	-	-	-	-	**96**

FIGURE A.1 – Source : https ://www.iot-lab.info/deployment/

A.2 Sortie de Massif

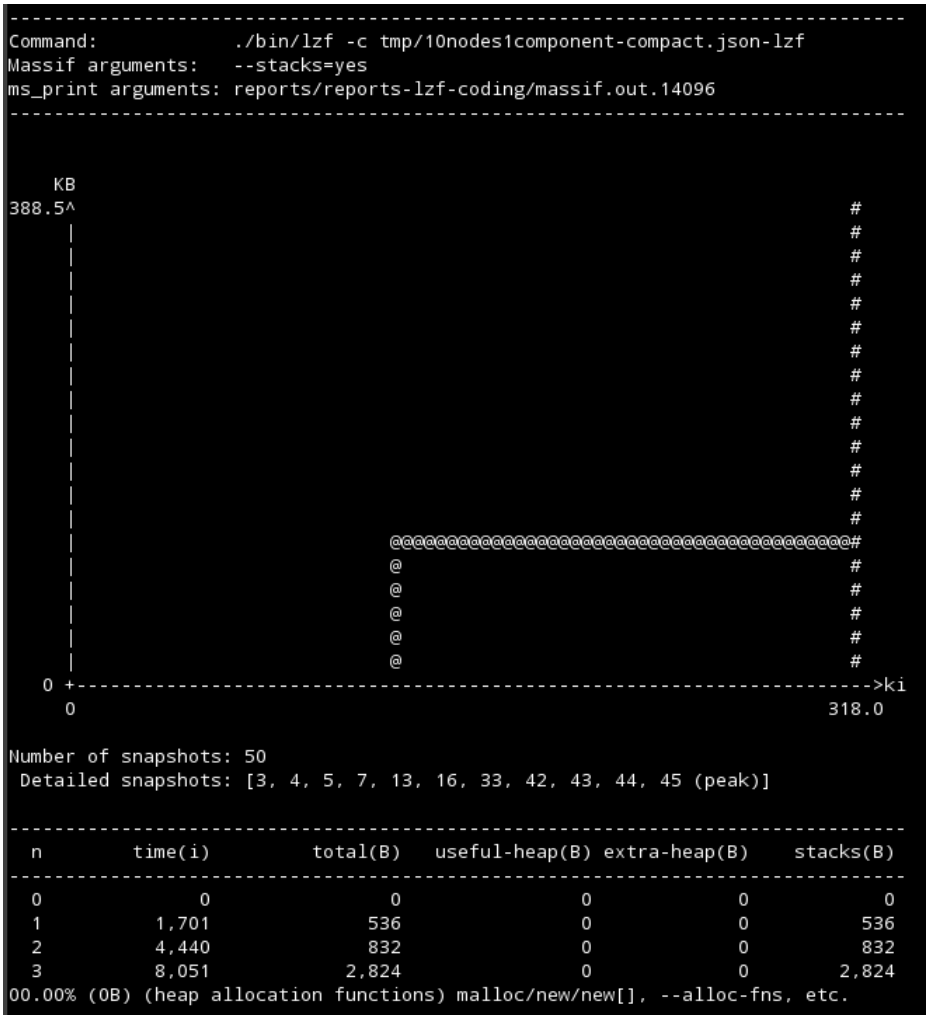


FIGURE A.2 – Exemple d'extrait de sortie de Massif, ici l'utilisation maximum de RAM est au snapshot 45 et est d'environ 400KB.

A.3 Compression de modèles

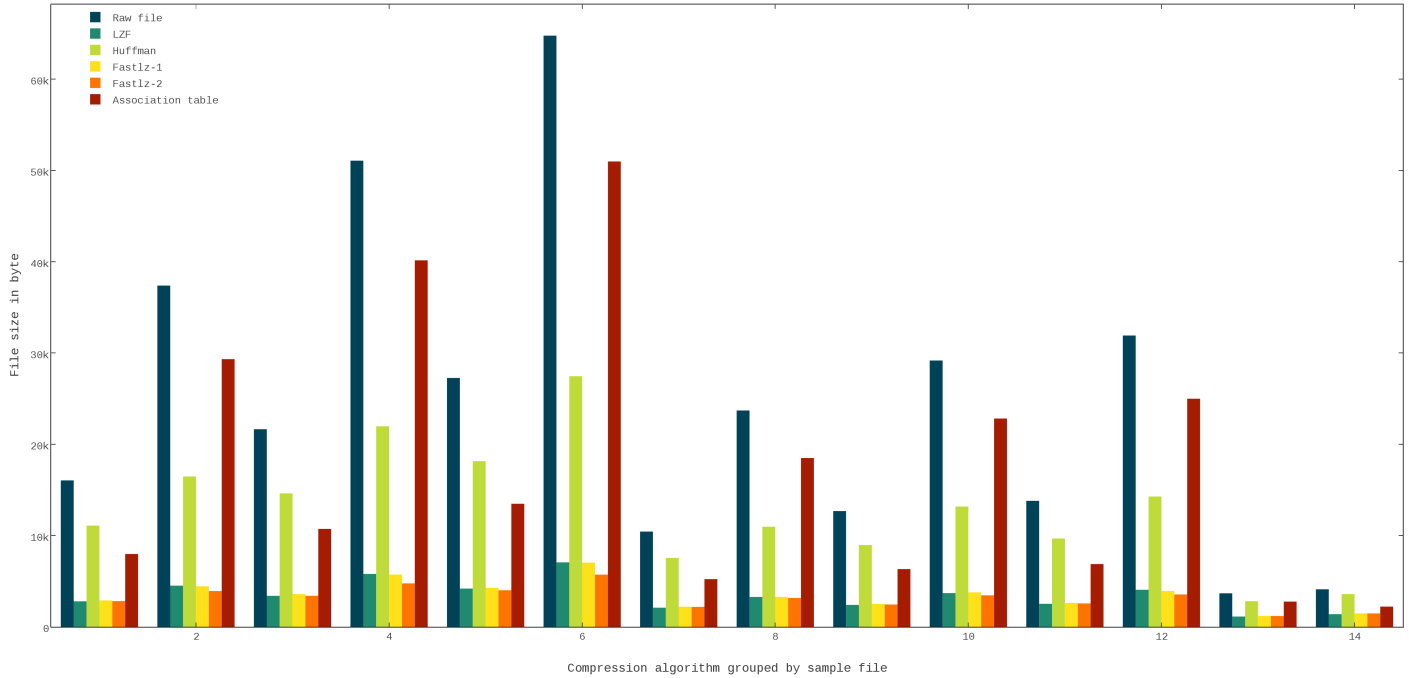


FIGURE A.3 – Mesure de compression de modèles à l'aide de différents algorithmes

A.4 Méta-modèle Kevoree

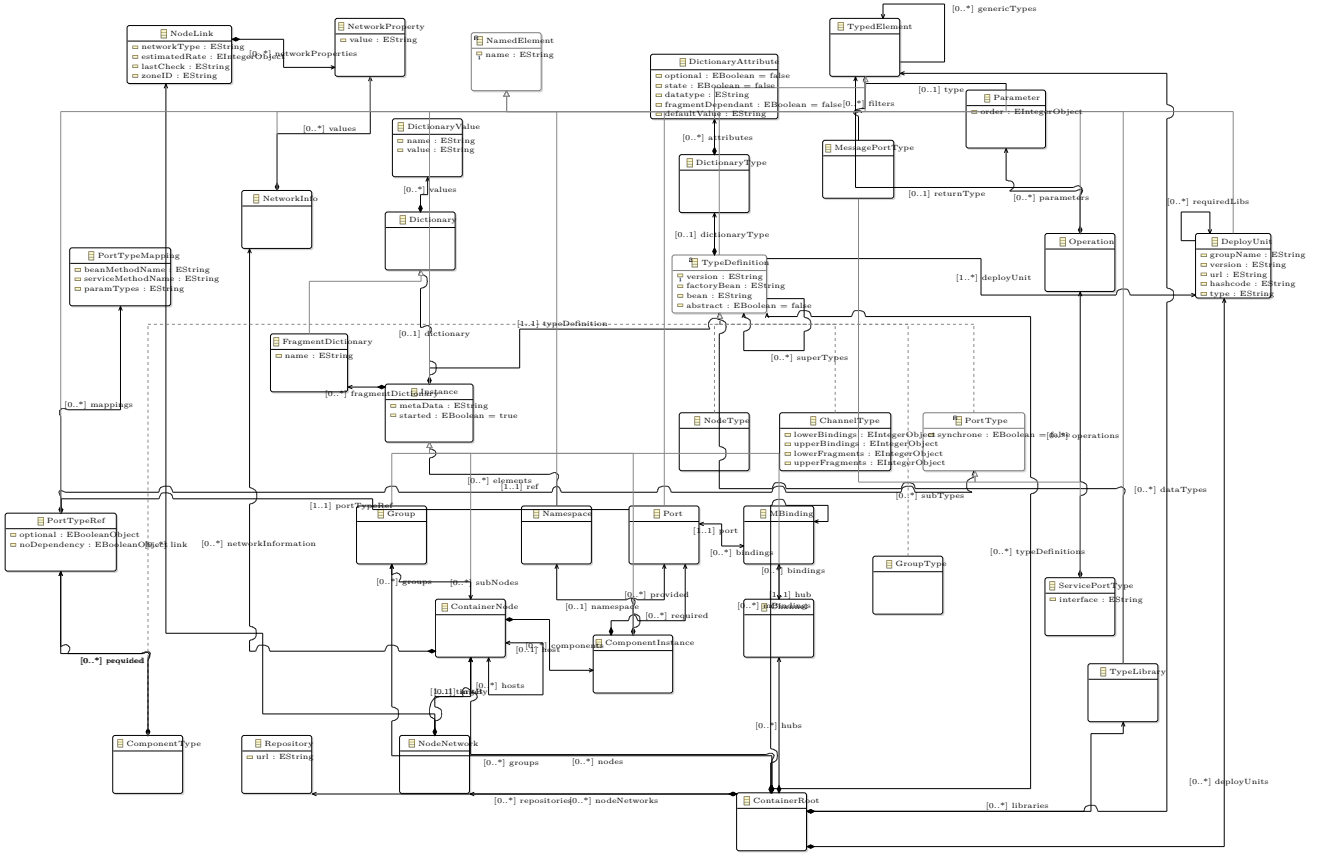


FIGURE A.4 – Représentation du méta-modèle de Kevoree.

A.5 Exemple de `struct` représentant une classe

```
1  typedef DictionaryValue* (*ftprDictionaryFindValuesByID)(Dictionary*, char↩
       *);
2  void initDictionary(Dictionary*);
3  Dictionary* new_Dictionary(void);
4
5  typedef struct _VT_Dictionary {
6      VT_KMFContainer *super;
7      /* KMFContainer */
8      fptrKMFMetaClassName metaClassName;
9      fptrKMFInternalGetKey internalGetKey;
10     fptrKMFGetPath getPath;
11     int (*fptrToJSON)(void*);
12     fptrFindByPath findByPath;
13     fptrDelete delete;
14     /*Dictionary*/
15     ftprDictionaryAddValues dictionaryAddValues;
16     ftprDictionaryRemoveValues dictionaryRemoveValues;
17     ftprDictionaryFindValuesByID dictionaryFindValuesByID;
18 } VT_Dictionary;
19
20 typedef struct _Dictionary {
21     VT_Dictionary *VT;
22     /* KMFContainer */
23     KMFContainer *eContainer;
24     /* Dictionary */
25     char generated_KMF_ID[9];
26     map_t values;
27 } Dictionary;
```

Listing A.1 – Extrait du fichier Dictionary.h.

Le *model@runtime* est un paradigme visant à permettre l'adaptation et la modification de réseaux de machines au travers de modèles. Si cela fonctionne déjà sur des réseaux de serveurs des thèses s'efforcent de démontrer que le cas est également possible pour l'Internet des Objets, où les machines sont bien plus limitées en terme de puissance de calcul et d'espace mémoire.

Ce stage s'inscrit dans une de ces thèses et proposent une implémentation d'un compilateur lisant un méta-modèle décrivant un programme implémentant le paradigme du *model@runtime* et produisant un programme équivalent pour fonctionner sur un réseau de capteurs.