Nicolas Matusiak
Alexis Jacquesson

RAPPORT PIDR

Outil de visualisation dynamiquepour la vérification d'applications distribuées dans SimGrid.

Encadrants : Martin Quinson et Marion Guthmuller

SOMMAIRE

Introduction **1**

Objectif du module 1

Contexte et presentation de l'équipe 1

Analyse du sujet **3**

Organisation **4**

Travail réalisé **5**

Description du travail effectué 5

Solutions envisagés 8

Difficultés 9

Améliorations envisageables **10**

Conclusion **11**

Glossaire **12**

Sources **13**

INTRODUCTION

Objectif du module :

Le Projet Interdisciplinaire ou Découverte de la Recherche est un projet de 6 mois permettant aux élèves de deuxième année de TELECOM Nancy de découvrir le monde de la recherche. Le but de ce projet est de développer chez les élèves leur réflexion autour de sujets complexes, cela permet aussi d'explorer différents domaines inconnus chez des étudiants ingénieurs. Le but est donc de trouver des solutions inconnues à partir d'une reflexion nouvelle ou guidée.

Contexte et Présentation de l'équipe :

Durant notre projet nous avons travaillé sur une application d'un framework appelé Simgrid. Simgrid est un framework de simulation développé par plusieurs équipes de recherche et qui a pour but de fournir un ensemble d'outils pour la simulation d'applications distribuées pour différents environnements (Cloud, Pair à Pair ...). Il est utilisé par des milliers de personnes dans le monde, ce qui fait de ce logiciel open source un leader dans son domaine.

Nous avons travaillé avec l'équipe Algorille dirigé par Martin Quinson, un de nos encadrant. Cette équipe de recherche a pour but d'améliorer l'efficacité de calcul des systèmes distribuées à travers deux axes de recherche : la gestion transparentes des ressources et la structuration des applications pour ce type de calcul.

INTRODUCTION

Au sein de cette équipe notre deuxième encadrante Marion Guthmuller s'occupe du model-checker de Simgrid. Il permet à des personnes de faire la vérification formelle d'applications. Le principe du modele-checking est de parcourir l'ensemble des exécutions possibles de l'application, que l'on nommera par la suite états, et de vérifier si cela satisfait aux propriétés données. Le but de cette application et de vérifier si un algorithme n'effectue pas de bouclage dans le déroulement de ses états. Aujourd'hui l'application peut simuler un algorithme qui se partage avec cinq machines différentes, mais il a pour objectif de pouvoir simuler plus d'ordinateurs. C'est dans ce cadre que notre projet s'inscrit, en effet l'objectif final de la visualisation dynamique du graphe doit permettre à l'utilisateur de interagir avec l'outil pour une meilleur utilisation des ressources disponible.

ANALYSE DU SUJET

Le but de notre projet a été de créer un outil de visualisation dynamique pour le model-checker et permettre une utilisation plus efficace de l'application. L'objectif final et de permettre une totale interactivité entre l'interface graphique Java et l'outil de Simgrid.

Au début du projet le modele-checker générait un graphe grâce à l'outil dot sous forme d'image PNG. Le problème de ce graphe et que dans le cas d'une application qui génère beaucoup d'exécution, on se retrouve avec une image avec pleins d'informations qui ne sont pas nécessaires à l'utilisateur. On peut citer l'exemple de certaines exécutions de l'application qui sont affichés et que l'on a pas besoin d'étudier. Pour améliorer la visualisation nous devions réaliser une application Java qui dans un premier temps nous donne le même résultat que celui obtenu précédemment. Les objectifs donnés étaient de faire évoluer la visualisation pour la rendre plus agréable pour et de permettre à l'utilisateur d'afficher plus de détails sur les états de l'application simulée sans que cela n'encombre la visualisation du graphe. Pour cela plusieurs méthodes pouvaient être envisagées, notamment la réduction de certaines branches du graphe dont l'exploration est inutile selon le jugement de l'utilisateur. Lorsque ces premiers points étaient traités, nous devions regarder si certaines informations pouvaient être récupérées par l'application et affichées selon la demande. Enfin si tout cela était terminé, nous pouvions nous attaquer à l'intéraction entre le graphe et le modele-checker, un des objectifs était de pouvoir cacher des branches de simulations entières à volonté.

Nous avions donc un sujet très dense qui nous laisser aussi beaucoup de liberté dans les choix de conception.

ORGANISATION

Notre projet s'est déroulé en deux phases, la première de Janvier à Mars où nous avons effectué une réunion par semaine avec nos encadrants et le travail de développement ensemble. Cette première période nous a permis de prendre en main les différentes technologies utilisées et de poser différentes questions pour obtenir l'ensemble des spécificités que notre interface devait fournir. Puis de Mars jusqu'à la fin, nous nous sommes organisé de la façon suivante : nous avons séparés le travail en deux , d'un côté la réalisation du parser et le parsage de la console pour afficher le graphe au fur et à mesure que l'application explore les différentes exécutions du programme, de l'autre côté le développement de l'interface graphique et la prise en main du framework JUNG.

Simgrid est un framework qui est disponible sur les distribution Linux Ubuntu et Debian, nous avons donc développé sur ces deux distributions.

Notre interface graphique est développé en Java, nous avons utilisé l'IDE eclipse pour nous aider à écrire le code. Enfin pour l'affichage du graphe en Java nous avons utilisés un framework appelé JUNG. Ce framework permet de créer des graphes mais il fournit aussi des outils pour créer une interface de visualisation plus dynamique que une simple image.

Enfin nous avons créé un dépôt git sur le site bitbucket.org pour pouvoir partagé nos fichiers et accéder au code afin que chacun puisse le modifier chacun de son côté

TRAVAIL REALISE

Description du travail effectué :

Notre travail s'est réalisé en deux parties, après avoir pris connaissance de la façon dont fonctionnait le framework Simgrid, nous avons parsé les différentes informations utiles qui sont affichées dans la console. Chaque ligne dans la console correspond à un état de l'application.

```
1  ./bugged1 ../msg_platform.xml deploy_bugged1.xml --cfg=model-check:1 --log=mc_dpor.thres:debug
```

Ligne de commande pour lancer le model-checker

```
1  [HostB:client:(2) 0.000000] [example/INFO] Sent!
2  [HostC:client:(3) 0.000000] [example/INFO] Sent!
3  [HostA:server:(1) 0.000000] [example/INFO] OK
4  [HostD:client:(4) 0.000000] [example/INFO] Sent!
5  [HostB:client:(2) 0.000000] [example/INFO] Sent!
6  [HostC:client:(3) 0.000000] [example/INFO] Sent!
7  [HostB:client:(2) 0.000000] [example/INFO] Sent!
8  [HostC:client:(3) 0.000000] [example/INFO] Sent!
9  [HostA:server:(1) 0.000000] [example/INFO] OK
10 [HostD:client:(4) 0.000000] [example/INFO] Sent!
11 [HostB:client:(2) 0.000000] [example/INFO] Sent!
12 [HostC:client:(3) 0.000000] [example/INFO] Sent!
13 [HostB:client:(2) 0.000000] [example/INFO] Sent!
14 [HostA:server:(1) 0.000000] [mc_global/INFO] *************************
15 [HostA:server:(1) 0.000000] [mc_global/INFO] *** PROPERTY NOT VALID ***
16 [HostA:server:(1) 0.000000] [mc_global/INFO] *************************
17 [HostA:server:(1) 0.000000] [mc_global/INFO] Counter-example execution trace:
18 [HostA:server:(1) 0.000000] [mc_global/INFO] [(1)HostA (server)] iRecv(dst=(1)HostA (server), buff=(verbose only), size=(verbose only)) (56)
19 [HostA:server:(1) 0.000000] [mc_global/INFO] [(2)HostB (client)] iSend(src=(2)HostB (client), buff=(verbose only), size=(verbose only)) (54)
20 [HostA:server:(1) 0.000000] [mc_global/INFO] [(1)HostA (server)] Wait(comm=(verbose only) [(2)HostB (client)-> (1)HostA (server)]) (62)
21 [HostA:server:(1) 0.000000] [mc_global/INFO] [(1)HostA (server)] iRecv(dst=(1)HostA (server), buff=(verbose only), size=(verbose only)) (56)
22 [HostA:server:(1) 0.000000] [mc_global/INFO] [(2)HostB (client)] Wait(comm=(verbose only) [(2)HostB (client)-> (1)HostA (server)]) (62)
23 [HostA:server:(1) 0.000000] [mc_global/INFO] [(4)HostD (client)] iSend(src=(4)HostD (client), buff=(verbose only), size=(verbose only)) (54)
24 [HostA:server:(1) 0.000000] [mc_global/INFO] [(1)HostA (server)] Wait(comm=(verbose only) [(4)HostD (client)-> (1)HostA (server)]) (62)
25 [HostA:server:(1) 0.000000] [mc_global/INFO] [(1)HostA (server)] iRecv(dst=(1)HostA (server), buff=(verbose only), size=(verbose only)) (56)
26 [HostA:server:(1) 0.000000] [mc_global/INFO] [(3)HostC (client)] iSend(src=(3)HostC (client), buff=(verbose only), size=(verbose only)) (54)
27 [HostA:server:(1) 0.000000] [mc_global/INFO] [(1)HostA (server)] Wait(comm=(verbose only) [(3)HostC (client)-> (1)HostA (server)]) (62)
28 [HostA:server:(1) 0.000000] [mc_global/INFO] Expanded states = 22
29 [HostA:server:(1) 0.000000] [mc_global/INFO] Visited states = 56
30 [HostA:server:(1) 0.000000] [mc_global/INFO] Executed transitions = 52
31 Aborted (core dumped)
32
```

Informations obtenues dans la console

Le parser qui a été retenu au final fonctionne grace aux expressions régulières. Lorsque Simgrid est lancé, il renvoie en console une succession de lignes appellé "trace". A chaque ligne état correspond un format de ligne. On peut donc créer une expression régulière et récupérer les informations utiles à la construction du graphe, telles que la source, la destination etc… La trace est enregistré dans un fichier grâce à la fonction script en shell, c'est à partir de ce fichier que nous construisons le graphe en Java, nous choisissons le fichier grâce à une fenêtre qui apparait au lancement de l'application.

TRAVAIL REALISE

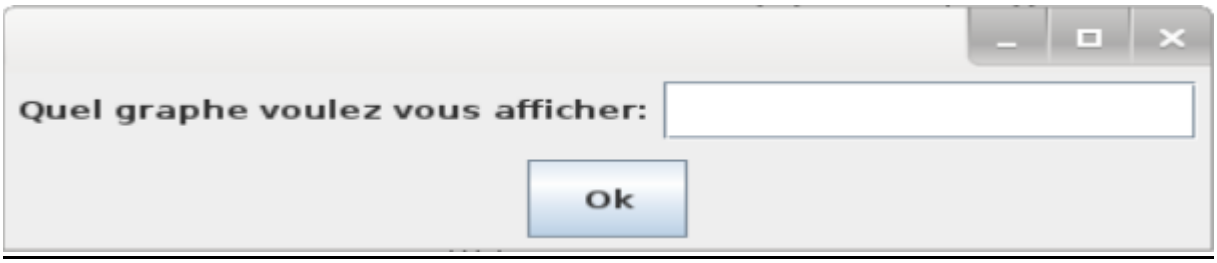
exemple :

mc_dpor.c:452: [mc_dpor/DEBUG] Execute: [(4)HostD (client)] Wait(comm=(verbose only) [(4)HostD (client)-> (1)HostA (server)]) (62)

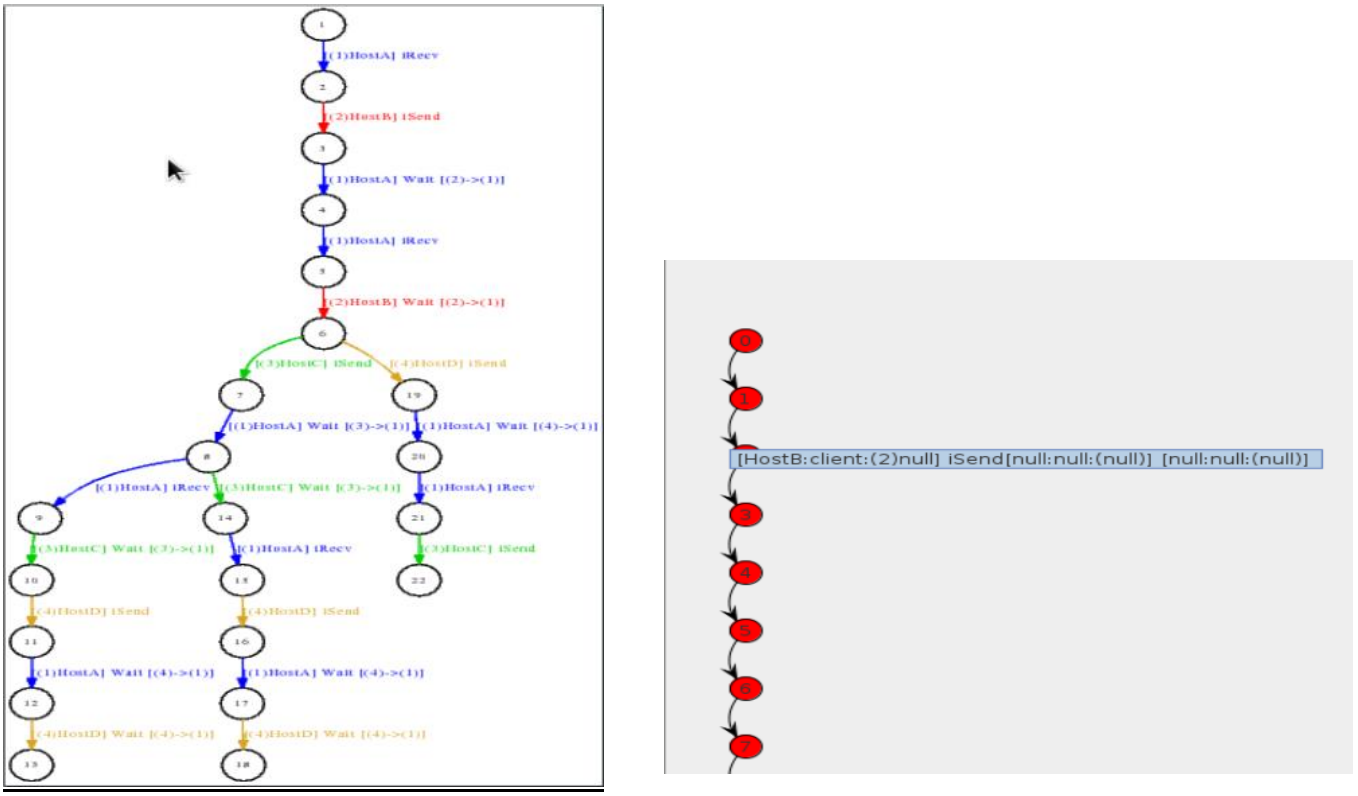
^[a-z_.]+:[0-9]+: [\\x5B][a-zA-Z_/]+[\\x5D] [a-zA-Z]+: [\\x5B][\\x28][0-9]+[\\x29][a-zA-Z]+ [\\x28][a-z]+[\\x29][\\x5D] [a-zA-Z]+[\\x28][a-z]+=[\\x28][a-z]+[\\x29] [\\x5B][\\x28]([0-9]+)[\\x29]([a-zA-Z]+) [\\x28]([a-z]+)[\\x29][\\x2D][\\x3E] [\\x28]([0-9]+)[\\x29]([a-zA-Z]+) [\\x28]([a-z]+)[\\x29][\\x5D][\\x29] .*$

Puis nous avons travaillé sur l'affichage du graphe, pour cela nous avons utilisé le framework JUNG. Les différentes informations récupérées grâce au parsage nous ont permis de construire les nœuds et les liens du graphe. Ce qui nous a permis d'obtenir le premier affichage demander, mais nous trouvions que les détails n'étaient pas très visible. Pour résoudre ce problème nous avons décider d'afficher les différentes informations dans des infobulles (ou tooltiptext), ce qui nous permet d'afficher le détail d'une exécution en passant la souris sur le liens qui correspond. On a aussi ajouté une fonction qui permet déplacer le graphe dans la fenêtre d'affichage, car dans le cas d'un grand graphe il ne peut pas être afficher dans toute la fenêtre. Par la suite nous avons envisagé plusieurs fonctions qui n'ont pas marché.

TRAVAIL REALISE



Fenêtre afficher au lancement de l'application Java



Comparaison entre le grahe dot et le graphe java

TRAVAIL REALISE

Solutions envisagés :

Tout d'abord nous voulions que l'application modele-checker de Simgrid puisse se lancer à partir de l'interface Java que nous avons développés. Puis nous avons essayés de parser directement la console et d'afficher le graphe en même temps que le model-checker soit lancer. Pour cela les deux applications devaient être lancer en même temps, mais nous n'avons pas réussi à lancer la ligne de commande dans le terminal à partir de Java. Puis nous nous sommes consacrés à l'affichage du graphe et à ses différentes options, comme le fait de cacher certains états du graphe et bien sûr la suite du graphe qui dépend de cet état. Pour cela nous avons créer un menu qui s'affiche lorsqu' on clique sur le noeud du graphe, ce menu contient un bouton qui permet de cacher l'état, mais pour cacher les états suivant cela ne marche pas.

Même si nous avons commencé plusieurs fonctions pour ce projet, finalement peu on aboutie à cause des difficultés qu'on a eu avec le framework JUNG.

TRAVAIL REALISE

Difficultés :

Au tout début, nous avons passé un certains temps sur le parser de notre application. La difficulté venait du fait qu'il a été impossible d'utiliser les caractères spéciaux dans les fonctions spécifiques au parsage, nous avons donc utilisé le code hexadécimale des caractères spéciaux dans les expressions régulières.

Les problèmes que nous avons rencontrés par la suite dans les différentes fonctions de notre projet sont liés d'une part avec le framework JUNG, mais aussi le problème de lancement d'une commande en Java.

Pour la partie concernant l'affichage avec JUNG, nous avons eu des soucis à modifier les nœuds et les liens pour en faire ce que nous voulions. En effet le problème du framework se situe qu'il est difficile d'intégrer plusieurs fonctions qui intéragissent avec les noeuds et les liens du graphe. Notre application veut aussi intérargir avec des noeuds et des liens précis et l'accessibilité au éléments graphiques avec jung n'est pas un chose aisé. De ce fait l'intégration de fonction qui ne sont pas intégrer au framework ne sont pas possibes. On peut donc conclure sur le fait que notre avancement sur la partie de visualisation dynamique ne ressemble pas à ce que nous avions penser au départ.

AMELIORATIONS ENVISAGEABLES

Les améliorations a apporté à ce projet sont l'affichage dynamique du graphe et l'intéraction graphique. L'objectif final étant d'avoir une interface graphique qui puisse gérer un gros volume d'éléments et de les afficher sur commande. Or actuellement tout le graphe est affiché d'un coup. Mais on a aussi besoin d'une application qui puisse lancer le model-checker en utilisant les fichiers exemples de Simgrid comme arguments, actuellement tout est fait "à la main" c'est à dire étape par étape, on ne l'a pas automatisé.

On peut également envisager la création d'une liste de branche amenant à une erreure et une liste de branche amenant à un résultat cohérent, que l'on sauvegarderait quelque part grace à un identifiant, ces listes se créant en même temps que le graphe. Ces listes fourniraient déjà une bonne information pour l'utilisateur. Ensuite on pourrait rajouter un raccourci pour mener vers l'étape "source" de la branche en question.

CONCLUSION

La découverte de Simgrid a d'abord été impressionnante pour nous deux puis est devenu un frein à cause du temps de mise en place. De manière générale suivre un calendrier précis et répartir les taches pour mener à bien notre projet s'est révélé ardue. Cependant nous parvenons à offrir une interface et une application stable et modulable pour visualiser la trace donnée par Simgrid.

L'interface idéale serait une représentation qui se crée au fur et à mesure de la lecture de la trace, mais les difficultés rencontrées nous ont empêché de terminer à temps une telle interface, bien que nous soyons convaincus que l'objectif puisse etre atteint.

Notre représentation est crée en une fois et nous avons ensuite la possibilité de voyager et d'obtenir plus de détails sur les états.

Au final nous avons découvert un projet ambitieux, complexe et long de plusieurs années, sur lequel nous avons tentés d'apporter une contribution qui puisse rendre service à l'équipe de recherche.

GLOSSAIRE

- Simgrid :
  Outil de simulation pour application distribuée.

- Model-checker :
  Outil appliquant les techniques de model-checking, cela permet d'effectuer des vérifications automatiques pour les systèmes dynamiques.

- JUNG :
  Java Universal Network/Graph

- IDE :
  Integrated Development Environment ou Environnement de développement.

SOURCES

Pour le parser :

http://www.regexr.com/

Pour Simgrid :

http://simgrid.gforge.inria.fr/simgrid/latest/doc/

http://simgrid.gforge.inria.fr/tutorials/simgrid-101.pdf

http://simgrid.gforge.inria.fr/tutorials/simgrid-mc-101.pdf

Pour Jung :

http://sourceforge.net/apps/trac/jung/wiki/JUNGManual

http://www.grotto-networking.com/JUNG/JUNG2-Tutorial.pdf

http://jung.sourceforge.net/doc/JUNGVisualizationGuide.html

http://jung.sourceforge.net/doc/api/index.html