

- Position Paper -
OSMOSE :
Outil d'analyse de codes binaires,
application au domaine de l'énergie. *

S. Bardin P. Herrmann
CEA,LIST, Laboratoire de Sécurité logicielle
Boîte 65, Gif-sur-Yvette, F-91191 France
sebastien.bardin@cea.fr
philippe.herrmann@cea.fr

A. Ourghanlian N. Richer
EDF Recherche & Développement
Chatou, 78401, France
alain-1.ourghanlian@edf.fr
nicolas.richer@edf.fr

9 mars 2007

1 Introduction

L'introduction de plus en plus importante de logiciel dans de nombreux composants des systèmes de contrôle commande, en particulier des systèmes supports de fonctions pouvant avoir un *impact sur la sûreté*¹ des installations de production d'énergie, conduit EDF à se doter d'approches outillées permettant d'évaluer ces logiciels par une analyse de leur code source ou le cas échéant par une analyse de leur code binaire.

Les outils actuellement disponibles sur le marché portent essentiellement sur une analyse du code source de haut niveau, de type langage C ou Ada. Toutefois pour certains logiciels, une analyse au niveau du code assembleur ou binaire est nécessaire ou pourrait apporter un plus. Ce pour trois raisons :

1. Les logiciels développés en langage de haut niveau comme le langage C, ont des portions écrites directement en assembleur (logiciel proche du matériel), voire ont été complètement programmés en assembleur lorsqu'il s'agit de systèmes développés il y a plus de vingt ans et toujours opérationnels.
2. Certaines vérifications de propriétés ne peuvent être faites qu'au niveau du code assembleur ou binaire, comme l'évaluation du pire temps d'exécution ou l'estimation de la taille de pile utilisée par une tâche afin de s'assurer que le dimensionnement est correct.
3. Certains composants sont développés non spécifiquement pour le domaine nucléaire. Pour des raisons de propriétés industrielles et de faible volume commercial, les fournisseurs de tels composants ne nous permettent pas d'accéder au code source des logiciels embarqués.

L'outil OSMOSE. Dans ce contexte, EDF est intéressé par le développement de nouveaux outils capables de travailler au niveau du code binaire, c'est à dire au plus près du logiciel réel. Ces outils doivent dans un premier temps construire un modèle du logiciel à partir de son code binaire exécutable. Il est alors possible dans un second temps d'exploiter ce modèle avec différentes techniques d'analyse. L'outil OSMOSE est développé par le CEA LIST, en partenariat avec EDF R&D. L'outil vise à automatiser au maximum l'analyse de codes binaires, en se concentrant sur les deux points suivants :

*Ce travail fait partie du projet *Usine Logicielle*, financé en partie par le pôle de compétitivité SYSTEM@TIC PARIS-REGION.

¹Par impact sur la sûreté, nous désignons les impacts possibles pour la sûreté des personnes, des biens ou de l'environnement.

- Rétro-conception² : aider l'utilisateur à comprendre plus rapidement le fonctionnement du programme en lui fournissant un modèle de haut niveau du code (flot de contrôle, graphe d'appel, etc.).
- Génération de test : générer automatiquement un jeu de tests avec un objectif de couverture structurel (couverture des instructions ou des branches) et une estimation du niveau de couverture obtenu.

Plan du papier. Avant de présenter l'outil, la section 2 fait le point sur les spécificités des codes binaires et des logiciels embarqués, pour bien mettre en évidence les problèmes auxquels nous sommes confrontés. La section 3 présente les objectifs à moyen terme de l'outil OSMOSE, du point de vue utilisateur. Les technologies sous-jacentes et l'architecture du logiciel sont évoquées à la section 4. Un panorama des travaux apparentés est dressé à la section 5. Enfin la section 6 décrit l'état d'avancement de l'outil fin mars 2007 et le planning prévu pour 2007.

2 Spécificités du code binaire embarqué

Par rapport à une analyse statique classique sur un code source de haut niveau de type C ou Ada, nous partons du code binaire et les programmes sont destinés à être embarqués dans des systèmes réactifs. Cela engendre des problèmes bien spécifiques listés ci-dessous.

Différences entre codes binaires et langages de haut niveau. Un code binaire exécutable se présente comme une suite de bits. Ceci implique qu'il n'y a pas de boucles définies syntaxiquement, pas de fonctions, pas de types et même pas de variables. Ces concepts n'existent tout simplement pas à ce niveau de programmation. En fait, on ne connaît même pas le nombre ni la nature des instructions du programme : les instructions n'ont pas toujours la même taille, elles ne sont pas forcément alignées et peuvent potentiellement se chevaucher, on ne peut pas distinguer une instruction d'une donnée et finalement les sauts calculés³ peuvent a priori mener n'importe où dans le code.

Systèmes réactifs. Un système réactif interagit avec son environnement via des capteurs (prise d'information sur l'environnement) et des actionneurs (action sur l'environnement). Il faut donc modéliser cet environnement et son évolution, et prendre en compte que les variables correspondant aux lectures de capteurs sont volatiles. Pour ces systèmes, un cas de test est composé d'une valeur initiale pour chaque variable d'entrée et d'une séquence (plus ou moins longue) des valeurs lues sur chaque capteur.

Systèmes embarqués. La programmation embarquée utilise beaucoup d'instructions de bas niveau des processeurs, par exemple les opérations bit à bit (rotations, tests d'overflow, etc.) et la gestion explicite des interruptions. Ces opérations nécessitent une modélisation plus fine que les abstractions utilisées habituellement pour analyser les langages de haut niveau, comme par exemple prendre en compte que l'arithmétique des ordinateurs n'est pas idéale mais *modulo*⁴.

Récapitulatif des difficultés spécifiques. Nous listons les problèmes auxquels nous sommes confrontés. Nous avons ajouté une exigence destinée à élargir l'usage potentiel de l'outil.

1. Nous ne pouvons nous appuyer sur aucune information de haut niveau du programme telle que boucles, fonctions ou encore variables. Nous ne connaissons même pas à l'avance toutes les instructions du programme.
2. Le programme n'est pas isolé, mais communique avec un environnement. Il reçoit des informations via des capteurs et agit sur l'environnement via des actionneurs.
3. Nous devons prendre en compte des opérations de bas niveau : opérations sur les bits, interruptions, etc.
4. Enfin l'outil doit être le plus indépendant possible d'un jeu d'instructions donné, de manière à ce qu'il soit possible d'adapter un nouveau jeu d'instructions à coût raisonnable et sans passer par l'intermédiaire des développeurs de l'outil.

²Compréhension du fonctionnement, revue de conception, etc.

³Un saut calculé est une instruction `goto` dont l'adresse de branchement n'est connue qu'à l'exécution, et peut varier d'une exécution à l'autre.

⁴Ainsi, par exemple, pour des entiers non signés codés sur 32 bits, l'opération $4294967295 + 1$ retourne 0.

Certains de ces problèmes ne sont pas couverts par l'état de l'art, notamment la gestion des interruptions et la reconstruction d'un modèle de haut niveau du programme. Dans le contexte pratique industriel d'EDF, la résolution de certains des points précédents peut être rendue plus facile que dans le cas général, par exemple en prévoyant que l'utilisateur puisse définir le mapping mémoire instructions/données ou les cibles de certains sauts calculés. Cependant les analyses proposées se doivent tout de même d'être suffisamment précises, sous peine de noyer l'utilisateur sous un flot de demandes d'informations complémentaires.

3 Objectifs à moyen terme du projet OSMOSE

Cette section présente les objectifs à moyen terme du projet. Pour l'état d'avancement actuel, voir la section 6.

L'outil OSMOSE permet de répondre au besoin d'évaluation des logiciels à partir de leur code binaire. La figure 1 présente le contexte général d'utilisation de cet outil.

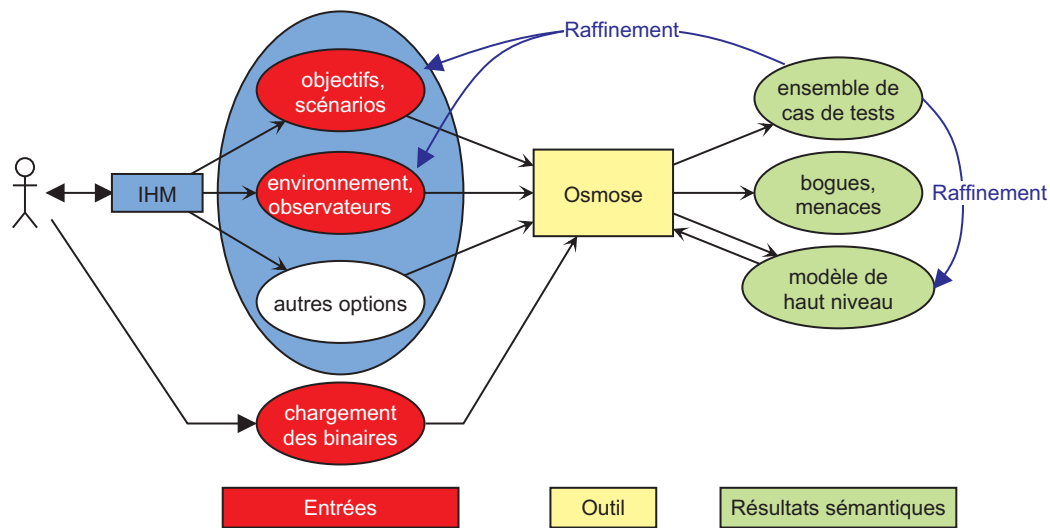


FIG. 1 – Schéma d'utilisation d'OSMOSE.

OSMOSE prend en entrée au moins un code binaire exécutable, le jeu d'instructions à utiliser⁵ et éventuellement une modélisation de l'environnement. Il fournit en sortie :

- Un modèle de haut niveau du programme pour faciliter sa compréhension.
- Un ensemble de cas de tests pour du test structurel, et une estimation de la couverture obtenue.
- Un rapport des bogues et des menaces découverts.

On entend par *bogues* des erreurs certaines (crash à l'exécution⁶) ou des violations de la politique de programmation⁷. Chaque bogue est accompagné d'un cas de test qui active le bogue. On désigne par *menaces* des points du code délicats sur lesquels l'outil n'a pas pu conclure. L'équipe de tests devrait donc examiner en priorité ces instructions. Des exemples de menaces sont les sauts calculés non résolus⁸ ou les instructions non couvertes par le jeu de tests.

⁵L'outil supportera dans un premier temps les jeux d'instructions pour PowerPC 555, M6800 et 8051.

⁶Par exemple, division par 0 ou accès mémoire illégal.

⁷L'utilisateur peut définir la politique souhaitée.

⁸Un saut calculé est résolu quand toutes ces cibles potentielles ont été découvertes et explorées.

4 Technologies et Architecture

OSMOSE prend en entrée un code exécutable et fournit en sortie une représentation de haut niveau du code, un ensemble de cas de tests et un rapport de l'exécution de ces tests. L'architecture est représentée dans la figure 2.

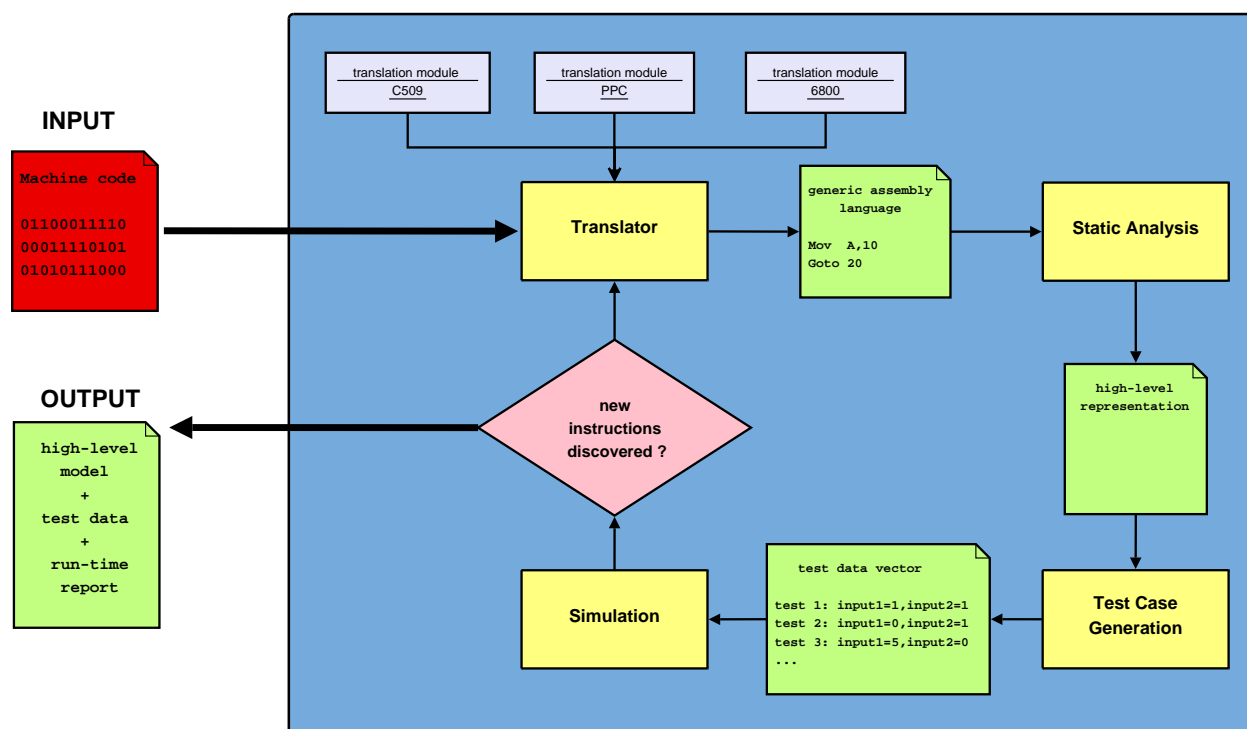


FIG. 2 – Architecture d'OSMOSE.

Pour assurer l'indépendance de l'outil vis-à-vis d'une cible spécifique, nous travaillons en interne sur un *langage assembleur générique*. L'exécutable est donc d'abord traduit dans notre assembleur générique par un module dédié. Seul ce module dédié doit être réécrit pour intégrer un nouveau jeu d'instructions. Puis nous utilisons une combinaison d'analyses statiques et dynamiques : une première analyse statique synthétise un premier modèle de haut niveau. Ce modèle est passé au *moteur de génération de tests* qui retourne un ensemble de cas de tests. Chacun des cas de tests est lancé en *simulation*. Quand de nouvelles parties du code sont découvertes, le modèle du code est mis à jour et le processus est relancé.

Quelles garanties ? Si l'outil a résolu tous les sauts calculés, alors le modèle est bien une sur-approximation du graphe de contrôle du programme et la couverture estimée pour le jeu de tests est inférieure à la couverture réellement atteinte. Dans le cas contraire⁹, le modèle est une sous-approximation et la couverture estimée est trop optimiste. Les bogues rapportés sont des erreurs *en simulation*. Ils doivent encore être validés sur le système réel, mais on s'attend avec une très forte probabilité à ce qu'ils correspondent à de vrais bogues. Les menaces rapportées sont potentiellement des fausses alarmes, car les technologies employées ne permettent pas, par exemple, de prouver qu'une branche est morte. Le but est ici d'attirer l'attention du testeur sur un point difficile du code, à lui ensuite d'aviser.

⁹Il est possible d'indiquer à l'utilisateur dans quel cas on se trouve.

5 Travaux apparentés

Génération de cas de tests. Les techniques de génération automatique de tests travaillent généralement à partir de spécifications de haut niveau et visent le test fonctionnel (*Model-based testing*). Les techniques de génération de tests structurels basées sur les programmes sources sont récentes [3, 5, 6]. Les différents travaux cités présentent des techniques similaires dont nous nous sommes inspirées. Cependant, comme nous travaillons sur des exécutables, nous avons dû adapter ces techniques, notamment car nous n'avons pas d'information de haut niveau sur le code et nous devons gérer des opérations bit à bit, plus délicates à modéliser que les instructions de plus haut niveau considérées dans les langages de programmation.

Analyse statique. Des travaux récents d'analyse statique [1, 2] visent à synthétiser des modèles de haut niveau (appelés ici IRs) à partir d'exécutables. Les techniques développées sont plus complexes que les nôtres, car les auteurs restent dans un cadre purement statique. La combinaison d'analyses statiques et dynamiques nous permet d'utiliser des analyses plus légères. Il faudrait étudier plus précisément les mérites réciproques des deux méthodes. Les mêmes auteurs (et quelques autres) ont également proposé dans [4] une méthode de vérification de binaires, en traduisant les IRs obtenus dans une extension d'automates à pile (WPDS) puis en utilisant des algorithmes de model checking sur les WPDS. Cependant les auteurs n'ont pas encore fourni de résultats conséquents d'expérimentations.

6 État d'avancement et Perspectives

État d'avancement fin mars 2007. Le prototype OSMOSE fonctionne sur de petits programmes jouets et permet effectivement de récupérer un modèle de haut niveau du code et de générer un ensemble de cas de tests. L'outil est de plus indépendant d'une architecture cible spécifique. Le jeu d'instructions pour 8051 est implanté, celui pour PowerPC 555 en passe de l'être. Les limitations actuelles sont principalement la non prise en compte de l'environnement, des interruptions, de certaines instructions bit à bit et de toutes les instructions sur les flottants. L'outil souffre également d'un manque d'évaluation et de l'absence d'IHM.

Évaluations à venir. Dans le cadre de MoDriVal, l'outil OSMOSE sera évalué en parallèle par EDF et Hispano-Suiza¹⁰. EDF utilisera l'outil sur un logiciel réactif de contrôle-commande pour cible M6800. Hispano-Suiza évaluera l'outil sur un calculateur de type FADEC¹¹ sur cible PowerPC 555. Enfin la simplicité d'intégration d'une nouvelle architecture cible sera démontrée par EDF, dont l'équipe plantera la cible M6800.

Perspectives 2007. L'objectif fin 2007 est d'avoir un outil capable d'analyser les binaires pour les architectures 8051, PowerPC 555 et M6800 et prenant en compte toutes les instructions des études de cas fournies par EDF et Hispano-Suiza. L'outil aura cependant les limitations suivantes : pas de traitement des flottants, pas de traitement des interruptions, pas d'IHM, potentiellement des problèmes de passage à l'échelle. Ces limitations feront l'objet de travaux ultérieurs. Le planning des développements prévus pour 2007 dans le cadre de MoDriVal est le suivant :

avril 2007	traitement de toutes les instructions du cas d'étude EDF implantation du module de traduction PowerPC 555 modélisation de l'environnement
juin 2007	début des évaluations sur les cas d'étude d'EDF et d'Hispano-Suiza
juin-novembre 2007	implantation du module de traduction M6800 réglages de l'outil selon le retour utilisateur correction des bogues découverts

¹⁰Plus d'informations à <http://www.hispano-suiza-sa.com/>

¹¹Organe de régulation du moteur, et l'un des plus gros ECU - Electronic Control Unit - que l'on trouve à bord des avions.

Références

- [1] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *CC'2004*. LNCS 2988. Springer, 2004.
- [2] G. Balakrishnan, T. Reps, D. Melski and T. Teitelbaum. WYSINWYX : What You See Is Not What You eXecute. In Proc. *IFIP Working Conference on Verified Software : Theories, Tools, Experiments*. 2005.
- [3] P. Godefroid, N. Klarlund and K. Sen. DART : Directed Automated Random Testing. In *PLDI'2005*. ACM, 2005.
- [4] T. Reps, S. Schwoon, S. Jha and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. In *Science of Computer Programming*, october 2005.
- [5] K. Sen, D. Marinov, and G. Agha. CUTE : A Concolic Unit Testing Engine for C. In *ESEC/FSE'2005*. ACM SIGSOFT Software Engineering Notes, volume 30. ACM, 2005.
- [6] N. Williams, B. Marre and P. Mouy. Interleaving Static and Dynamic Analyses to Generate Path Tests for C Functions. 3rd Workshop on System Testing and Validation (STV'2004).