Liste des projets

Branche BMC-SAT : https://github.com/Dowsers/Dowsers_Solution/tree/ak/bmc

Branche SMARTPULSE : https://github.com/Dowsers/Dowsers_Solution/tree/ak/smartpulse

Branche HLL-SMT2 : https://github.com/ssie2025-pfe2/hll2smt-py

1 Bounded model checking par le SAT

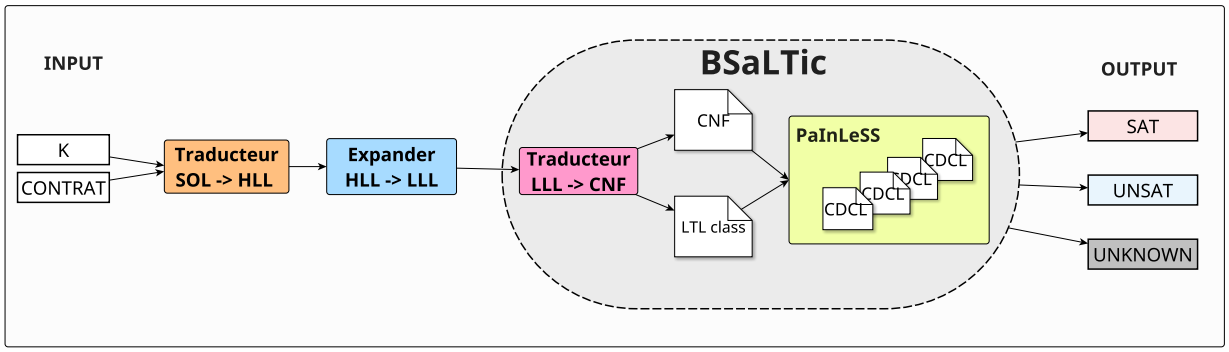

Figure 1: BMC-SAT architecture

1.1 Contexte

Le model checking [**?**] peut fournir un contre-exemple (CEX) lorsqu'un modèle ne satisfait pas une exigence (spécification). Ce CEX correspond à un chemin d'exécution du système, ce qui aide considérablement les concepteurs à comprendre où se situe le problème dans le système. Pour réaliser cette vérification, il est nécessaire d'effectuer une traversée complète de l'espace des états représentant les comportements du modèle. Deux approches ont été utilisées : le model-checking explicite et le model-checking symbolique.

Sur ce projet, nous nous concentrons sur les techniques symboliques qui utilisent des procédures de résolution SAT, devenues un pilier du model-checking moderne.

En particulier, les procédures SAT sont largement utilisées dans la version *bornée* du model-checking, notamment pour la vérification des spécifications LTL (Linear Temporal Logic).

Le Bounded Model Checking (BMC) [**?**, **?**, **?**] désigne une approche de model-checking où la vérification de la propriété est effectuée à l'aide d'une traversée limitée, c'est-à-dire une traversée d'une représentation symbolique de l'espace d'états qui est bornée par un entier k. Cette approche ne nécessite pas le stockage de l'ensemble de l'espace d'états, ce qui la rend plus souple et utile.

Les solveurs SAT modernes sont ainsi devenus la technologie centrale de nombreux model checkers. Cette Leurs efficacité est principalement due aux nombreuses optimisations qui ont été développées pour orienter les procédures SAT vers des espaces de recherche prometteurs, réduisant ainsi les temps de résolution. Une optimisation particulièrement remarquable implique la génération et l'utilisation d'informations de haute qualité (apprentissage) provenant des solveurs SAT, à base d'apprentissage par conflit [**?**, **?**] (Conflict Driven Clause Learning algorithm, CDCL), ce qui permet l'élagage de sous-espaces inutiles.

1.2 Branche BMC-SAT

– **Dépôt** : https://github.com/Dowsers/Dowsers_Solution/tree/ak/bmc

– **Objectif** : (1) Vérifier la fiabilité d'un Smart Contract sur une propriétés, passe par la résolution de la formule SAT qui les traduit. La vérification est limitée à nombre de transactions k défini au préalable.

(2) Étudier les informations importantes à sauvegarder lors de la phase d'apprentissage par conflit du SAT solveur CDCL. Ainsi, pouvoir adapter les heuristiques de stockage d'information selon les caractéristiques des contrats Solidity.

– **Entrée** : (1) Solidity Smart Contract version **0.8.x**,
(2) nombre de transactions k.

– **Sortie** : {CEX, OK, UNKNOWN} où CEX signifie que la propriété est invalide, OK elle est vérifiée et UNKNOWN si non décidé.

– **Langage pivot**: HLL

1.3 Calendrier

Avril-Juillet :

• Optimisation du convertisseur LLL vers le format d'entrée du SAT solveur (Forme Normale Conjonctive, CNF).

• Intégration d'un SAT solveur récent KISSAT-MAB [**?**] (gagnant de la compétition 2022).

• Adaptation des heuristiques de stockage d'information du SAT solveur pour les contrats Solidity.

• Expérimentation et comparaison de l'approche BMC-SAT avec l'existant (résolution par induction *PROVER*).

Octobre :

• Intégration complète de l'approche BMC-SAT avec le reste des techniques.

• Intégration d'une version concurrente des stratégies de vérification existantes (BMC-SAT et PROVER) : un portefeuille de stratégies qui résolvent le même contrat avec les mêmes propriétés (dépôt https://github.com/Dowsers/Tarkastus_Docker_AWS/tree/dev/portfolio)

Novembre-Décembre :

• Amélioration du projet : techniques de parallélisation, partage d'information précises, ...

2 Büchi contract

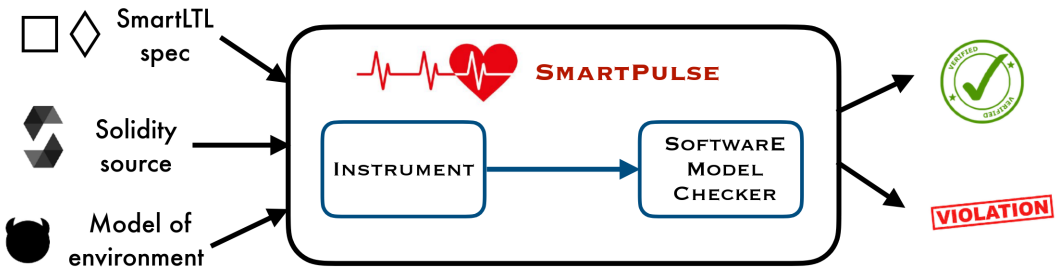

Figure 2: SmartPulse architecture

2.1 Contexte

Par Büchi contrat, nous entendons un smart contract accompagné de sa spécification représenté sous forme d'automate de Büchi.

Ce projet se focalise sur l'intégration de SmartPulse dans notre portefeuille de techniques de vérification. Produit par Microsoft employant des travaux de l'université de Freiburg, SmartPulse n'est plus maintenu depuis 2019.

L'outil repose sur trois principes de conception :

1. **Un langage de spécification ergonomique et expressif :** SmartPulse permet aux utilisateurs de spécifier leurs propriétés dans un langage appelé SmartLTL. SmartLTL est basé sur la logique temporelle linéaire (LTL), un formalisme (*intuitif*) pour exprimer les propriétés des traces dans le temps. Cependant, SmartLTL étend la LTL standard avec des constructions supplémentaires qui facilitent l'expression des propriétés de correction des smart contracts. SmartPulse est ainsi capable de vérifier les propriétés de vivacité (liveness).

2. **Possibilité de personnaliser les modèles d'attaque :** SmartPulse est paramétré par un modèle d'environnement qui permet d'expérimenter différents modèles d'attaque. Par exemple, l'approche permet aux utilisateurs de personnaliser les hypothèses sur la façon dont l'attaquant peut interagir avec un contrat par le biais d'appels externes.

3. **Automatisation et précision :** SmartPulse est un outil entièrement automatisé basé sur le paradigme du raffinement de l'abstraction guidé par les contre-exemples (CEGAR) et fournit une approche unifiée pour la recherche simultanée de preuves et de violations à l'aide d'un solveur SMT (SAT modulo Theory) [**?**]. En outre, dans les cas où SmartPulse signale une erreur, il peut générer une attaque concrète dans laquelle la propriété sera violée.

Partant d'une spécification formelle rédigée en SmartLTL, d'un contrat Solidity et d'un modèle d'attaque, SmartPulse effectue d'abord une séquence d'instrumentations de programme pour modéliser l'environnement d'exécution du contrat et utilise ensuite une approche de vérification basée sur CEGAR pour rechercher des violations de propriétés (au travers d'automate de Büchi).

2.2 Branche SMARTPULSE

– **Dépôt** : https://github.com/Dowsers/Dowsers_Solution/tree/ak/smartpulse

– **Objectif** : (1) S'approprier l'outil et le mettre à niveau (plusieurs expressions Solidity ne sont pas pris en compte : *inline assembly, bitwise operations, fonctions ABI, etc.*
(2) Intégrer la branche BMC-SAT dans SmartPulse.
(3) Implémenter des techniques de model-checking explicite (*emptiness check*).

– **Entrée** : (1) Smart Contract Solidity version **0.5.10**,
(2) Spécification au format SmartLTL,

(3) Model d'attaque *(optionnel)*.

– **Sortie** : {CEX, OK, UNKNOWN}

– **Langage pivot** : Boogie (*modifié*)

2.3 Calendrier

Septembre :

- Prise en main de l'outil *non documenté*.
- Difficultés d'installation de SmartPulse (packages obsolètes)
- Mise à niveau de certains sous-outils/packages (VeriSol [**?**], Corral [**?**]).

Octobre :

- Création d'un docker pour faciliter l'utilisation de SmartPulse.
- Temporairement et pour des objectifs d'expérimentation : création d'un script qui adapte des contrats Solidity **0.[5-8].x** en **0.5.10**.
- Génération automatique des spécifications ERC20 en SmartLTL.

Novembre-Décembre :

- Passage vers Solidity **0.8.x**
- Comparaison de SmartPulse avec les outils de vérification existantes (BMC-SAT et PROVER).
- Commencer l'optimisation des techniques de vérification de SmartPulse.

3 Convertisseur HLL vers SMT2

3.1 Contexte

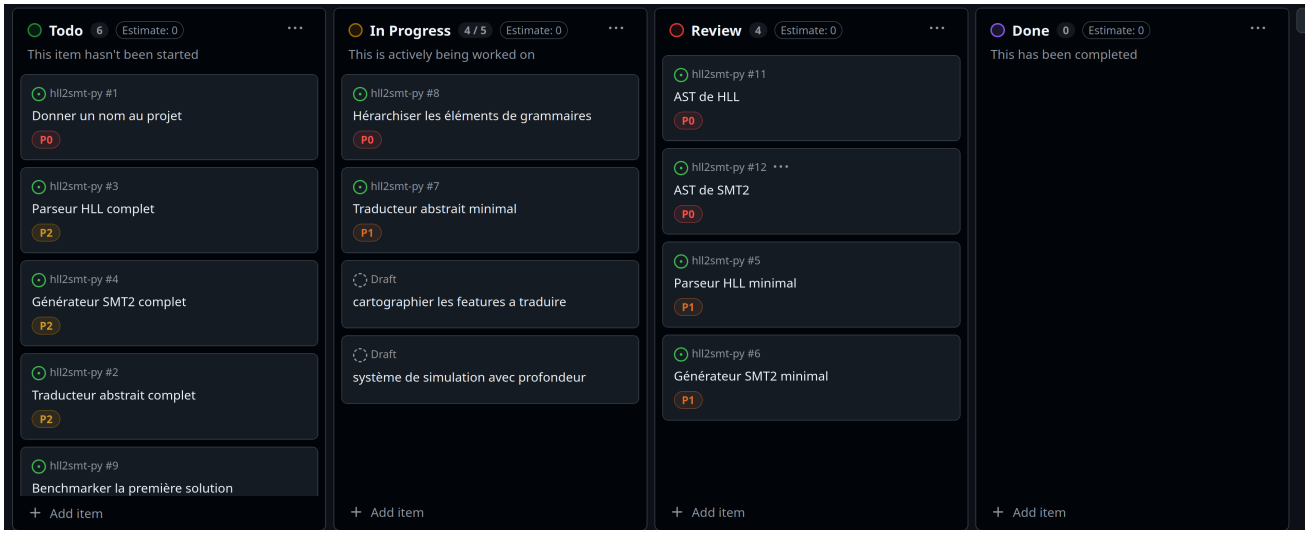

Figure 3: Avancement projet HLL-SMT2

Ce projet est en cours de développement par les étudiants de EPITA Toulouse. Il vise à construire un traducteur du langage pivot HLL vers un problème SMT2.

Les SAT Modulo Theory (SMT) [**?**] sont des problèmes de décision pour des formules de logique du premier ordre avec égalité (sans quantificateurs), combinées à des théories dans lesquelles sont exprimées certains symboles de prédicat et/ou certaines fonctions (*logique propositionnelle, arithmétique entiers et réels, bitvectors, arrays, dataType, floating points, string*).

3.2 Branche HLL-SMT2

– **Dépôt** : https://github.com/ssie2025-pfe2/hll2smt-py

– **Objectif** : (1) Traduire n'importe quelle instance HLL au format SMT2.
(2) Exploiter les approches de vérification formelle à base de solveur SMT.

– **Entrée** : (1) Fichier au format HLL,
(2) nombre de transactions k.

– **Sortie** : Fichier au format SMT2.

3.3 Calendrier

La figure 3 ci-dessus liste les avancés de ce projet.

Septembre-Octobre :
- Finalisation des Arbres Syntaxiques Abstraites (AST) du langage HLL et SMT2.
- Première conversion d'une instance HLL à une variable accomplie.
- Hiérarchiser les éléments de grammaires.
- Cartographie des syntaxes et expressions à traduire.

Novembre-Décembre :
- Introduire la notion de temporalité.
- Avoir un exemple complet fonctionnel.
- Appliquer sur de vrais Smart contracts.