Formalisation de la sémantique des appels système pour un système de type UNIX

Laurent Georget

Research Master Thesis

Research Master Thesis

Formalisation de la sémantique des appels système pour un système de type UNIX

Author :
Laurent Georget

Supervisors :
Mathieu Jaume
Guillaume Piolle
Frédéric Tronel
Valérie Viet Triem Tong
CIDRE

Résumé

Nous cherchons à tester `kblare`, un module de détection d'intrusion pour le noyau Linux. Les tests unitaires développés manuellement, pour lesquels testeur doit spécifier lui-même l'état de départ et l'état final attendu du système, ne donnent aucune garantie forte sur la couverture du code et sur la correction de `kblare`. Nous développons donc une approche pour la construction formelle de tests. En faisant l'hypothèse de travail que seuls les appels système causent des flux d'information dans un système d'exploitation, nous construisons une sémantique des appels système en termes de flux d'information nous permettant de définir de meilleurs tests. Nous montrons la faisabilité de cette approche en construisant une sémantique utilisant la logique transactionnelle concurrentielle. Nous présentons dans un second temps notre prototype d'interpréteur pour cette sémantique et montrons comment construire et exécuter de simples tests. Nous donnons nos premiers résultats. Ils sont positifs et nous ont convaincus du bien-fondé de l'approche, bien que nous nous heurtions à des problèmes de performance ainsi qu'à l'impossibilité de modéliser certaines situations. Nous concluons avec nos futures directions de recherche sur cette approche.

Mots-clé : Sécurité, Sémantique, Appels système UNIX, Test, Logique transactionnelle

Remerciements

De nombreuses personnes m'ont été grandement utiles et agréables au cours de mon stage de master II, je leur exprime ici toute ma gratitude.

En premier lieu, je remercie Mathieu Jaume, Guillaume Piolle, Frédéric Tronel, et Valérie Viet Triem Tong pour leur disponibilité, leur patience et les conseils précieux qui m'ont guidé tout au long de ce travail de recherche. Je remercie également Ludovic Mé, responsable de l'équipe CIDRE, et à travers lui toute l'équipe qui m'a accueillie pour mon stage.

Je remercie mes collègues stagiaires, ingénieurs, doctorants de la salle du fond du niveau 5 de Supélec dont on n'est toujours pas sûr du numéro pour leur bonne humeur, les discussions aussi passionnantes que variés que j'ai eues avec tous et leurs expériences diverses qu'ils n'ont pas hésité à partager avec moi (et entre autres, leur science du billard sur la pause du midi).

Je remercie enfin tous ceux qui ont contribué, de près ou de loin, à ce que je me retrouve à travailler dans ce qui m'intéresse le plus aujourd'hui. En particulier, je souhaite remercier Laurence Rozé, responsable de ma troisième année à l'INSA de Rennes , Éric Anquetil, responsable de ma quatrième année, Valérie Gouranton et Sandrine Blazy, responsables du master de recherche en informatique de l'université de Rennes I.

Tous ceux qui se sentiront oubliés ici sont priés d'accepter mes excuses et de se rajouter à la liste.

En ultime analyse, toute chose n'est connue
que parce que l'on veut croire la connaître.
— koân zen, anonyme

Table des matières

Liste des tableaux

1 Introduction

1.1 Mécanismes de protection de l'information

Très tôt, dans le domaine de la sécurité informatique sont apparus des mécanismes permettant de restreindre l'accès à certaines informations par des utilisateurs non autorisés. Un des premiers mécanismes est très primitif mais continue malgré tout d'être largement utilisé dans les systèmes d'exploitation aujourd'hui. Il consiste à attacher aux fichiers des *attributs* de protection et aux utilisateurs des *droits*. Les processus héritent leurs permissions de leur utilisateur propriétaire. Lorsque l'un d'entre eux veut ouvrir un fichier, si ses droits ne correspond pas aux attributs de sécurité du fichier, cela signifie que l'utilisateur n'a pas le droit d'*accéder* au fichier. L'accès en est donc refusé au processus. Ce mécanisme de sécurité est nommé *contrôle d'accès discrétionnaire* car les droits sont appliqués de manière spécifique à chaque élément du système. Cependant, ce mode de sécurité souffre de nombreuses failles. Le problème est mieux compris sur un exemple. Imaginons un téléphone portable dans lequel on cherche à contrôler qu'aucune donnée ne sort du carnet de contacts (premier conteneur de données) vers une *socket* réseau (autre conteneur de données). Ce contrôle devrait donc garantir donc qu'aucune information de contact n'est diffusée vers l'Internet. Cependant, comme on peut le voir même dans cet exemple très simple, il est nécessaire de distinguer l'information elle-même de ses conteneurs car on ne peut surveiller que les conteneurs, et non l'information elle-même. Par conséquent, on pourrait imaginer le scénario suivant : un service de sauvegarde chargé de faire une copie de sécurité du carnet de contact écrit toutes les informations de celui-ci dans un fichier temporaire, le temps de le copier sur un autre support. En même temps, un programme malveillant surveille l'apparition de ce fichier pour l'envoyer vers l'Internet. Dans ce cas, on voit que les information du carnet de contact ont finalement été divulguées, alors même que l'on contrôlait les communications entre le carnet de contact et l'Internet. Il est donc nécessaire de savoir à chaque instant de la vie du système dans quel conteneur se trouve l'information à protéger. Cela peut être réalisé à travers le *suivi de flux d'information*.

1.2 Suivi de flux dans les systèmes d'exploitation

Blare est un modèle de moniteur de suivi de flux d'information développé par l'équipe CIDRE[10, 5]. Il garantit des propriétés de confidentialité et d'intégrité sur les informations contenues dans les systèmes qu'il surveille en suivant les flux d'information entre des conteneurs d'informations. Pour résoudre le problème posé dans le paragraphe précédent, Blare, tout comme d'autres approches comme Flume[12] ou TaintDroid[7], utilise des *tags* pour suivre la propagation de l'information. Les tags sont des méta-informations attachées aux conteneurs qui peuvent servir à décrire deux choses : les informations effectivement contenues, d'une part, et les informations que ce conteneur est autorisé à contenir, d'autre part. Chaque fois que de l'information se déplace d'un conteneur à un autre au sein du système surveillé, les tags se propagent du conteneur d'origine vers celui de destination. Dans l'exemple précédent, lors de la copie de sauvegarde, le fichier temporaire se serait vu affecté les tags du carnet de contact duquel les informations étaient originaires. Par conséquent, lorsque le programme malintentionné aurait cherché à faire fuir le fichier sur l'Internet, le système aurait su qu'il y avait des informations en provenance du carnet de contacts dans le fichier, et donc il aurait pu empêcher l'envoi du fichier — ou du moins lever une alerte.

Comme on le constate, une partie du problème est donc à présent déplacé. Auparavant, le problème était que les informations pouvaient se trouver hors de leur conteneur d'origine, y compris

lorsque des actions valides — comme une copie de sauvegarde — étaient réalisées. Avec les tags, il est possible de suivre les informations lorsqu'elles changent de conteneur. À présent, le problème est d'affecter convenablement les tags aux conteneurs à l'initialisation du système et d'assurer leur propagation correcte. C'est sur ce point que notre travail de stage s'est concentré. En effet, la bonne propagation des tags est d'une importance critique pour la sécurité du système. Une mauvaise propagation équivaut à un mauvais suivi de la localisation de l'information et donc potentiellement à des risques de divulgation ou de corruption [1] indésirables.

1.3 Validation des moniteurs de flux d'information

Blare est en réalité un modèle. Il se décline en plusieurs implémentations : `kblare` pour le noyau Linux, `JBlare` pour la Java Virtual Machine et `AndroBlare` pour la plate-forme Android. Nous nous intéressons dans le cadre de ce travail à `kblare`. Pour valider le bon fonctionnement d'un moniteur de flux d'information, différentes approches sont possibles comme par exemple la preuve de programmes. Cependant, dans le cas de `kblare`, les approches formelles se révèlent difficiles à mettre en œuvre car il est très difficile de démontrer une quelconque propriété sur le code du noyau lui-même, du fait de sa taille et sa complexité. Nous avons donc choisi une approche plus réaliste, à savoir le test. Cependant, les jeux de test manuels ne garantissent pas réellement de propriétés très fortes de sécurité donc nous voulons construire les scénarios de test selon un critère formel, pour assurer une confiance proche de celle que nous donnerait une preuve formelle de la satisfaction de certaines propriétés du code.

Nous faisons l'hypothèse que tous les flux d'informations dans un système d'exploitation sont causés par des appels système. C'est une hypothèse raisonnable car dans un système d'exploitation, seul le noyau a accès aux périphériques matériels permettant de recevoir ou de stocker de l'information. De plus, les communications inter-processus nécessitent également généralement un ou plusieurs appels système pour mettre en place les structures permettant aux processus d'échanger de l'information. Ces structures sont les espaces de mémoire partagée, par exemple, ou encore les files de messages. Cette hypothèse a malgré tout quelques défauts, car elle oblige à surapproximer largement certains flux d'information. Par exemple, une fois que deux processus ont obtenu un espace de mémoire partagée, ils peuvent en principe échanger toutes les informations en leur possession sans appel système. Nous sommes donc obligés de considérer un flux intégral de chaque processus vers l'autre pour ne pas risquer d'en oublier, alors même qu'il est tout à fait possible que seule une variable ait été échangée, ou même que le flux ait en réalité eu lieu dans une seule direction.

Sous cette hypothèse — les appels système engendrent tous les flux d'information dans le système d'exploitation — nous pouvons considérer les processus du système comme l'environnement du noyau. Nous modélisons uniquement le noyau et considérons les processus environnants comme des structures opaques, utilisant l'interface que sont les appels système pour communiquer avec le noyau. Ainsi, nous adoptons une vue similaire à celle du système que nous voulons tester, le moniteur de flux d'information. En effet, `kblare` est implémenté comme un *patch* dans le noyau Linux. Il intercepte les actions du noyau pour en déduire les flux réalisés. Avec une sémantique formelle, en termes de flux d'information, de l'interface que les processus utilisent, c'est-à-dire les appels système, nous pourrions donc définir pour chaque appel système quels flux `kblare` devrait

1. Le problème de corruption, non évoqué dans l'exemple, peut subvenir si des informations provenant des différentes sources qui ne sont pas de la même qualité contribuent à un même contenu. Par exemple, une base de données reconstituée à partir de deux sauvegardes de dates différentes pourrait contenir des incohérences.

identifier. En conséquence, nous pourrions construire de manière formelle des jeux de tests portant précisément sur cette sémantique formelle.

1.4 Études liées

Étant donné l'importance critique des moniteurs de suivi de flux d'information et des moyens de sécurisation des systèmes en général, beaucoup d'approches ont été développées pour valider leurs propriétés. Parmi les différentes approches proposées figure la preuve formelle qui apporte les garanties les plus fortes sur le modèle de sécurité mais qui ne couvre pas toujours les implémentations. Des exemples de preuves sont celles de la démonstration sans assistant de preuve de la propriété de non-interférence du système d'exploitation sûr utilisant le contrôle de flux d'information Flume [11] ou encore la démonstration plus complète, et plus rigoureuse, de l'application d'une politique de flux d'information dans le système d'exploitation seL4 [15]. Ces travaux, quoique capitaux pour la conception des futurs systèmes d'exploitation et des moniteurs de suivi de flux d'information, souffrent de défauts majeurs. Premièrement, ils s'appliquent à des systèmes d'exploitation spécialement conçus et programmés à l'effet d'être démontrés et non à des systèmes disposant de nombreuses fonctionnalités, supportant beaucoup de matériel et largement distribués. Or, `kblare` a précisément été conçu pour le noyau Linux. Il est en fait codé en tant qu'une modification du noyau. Effectuer des preuves de corrections sur un système tel que celui-ci nécessiterait donc une preuve de correction du noyau tout entier, alors qu'il fait plusieurs millions de lignes de code et qu'il n'a jamais été programmé dans le but d'être prouvé. Deuxièmement, comme dit plus haut, ces preuves s'appliquent aux modèles et non aux implémentations elles-mêmes. Or, par le *refinement paradox* — le paradoxe de l'extension — la preuve qui est correcte pour le modèle peut ne pas couvrir l'implémentation qui en constitue une extension.

Une autre approche est la vérification de modèle (*model checking*). De nombreuses théories et outils ont été mis au point à ce jour pour vérifier la conformance d'un modèle de moniteur de suivi de flux d'information, muni de sa politique de sécurité, avec certaines propriétés de sécurité. L'étude de Zhao et Liu en 2013 en est un bon exemple [21]. Dans cet article, les auteurs proposent une méthode pour vérifier la configuration de moniteurs de suivi de flux d'information et alerter en cas d'erreur de configuration résultant en la perte des propriétés de confidentialité et d'intégrité par le moniteur. Cette méthode donne des garanties fortes mais elle ne prémunit pas des incorrections ou ambiguïtés dans l'implémentation. De plus, le moniteur lui-même n'est pas toujours aisément modélisable. Dans le cas de `kblare` qui est du code ajouté au noyau Linux par exemple, on pourrait imaginer de modéliser certaines parties comme les politiques de sécurité mais pas son exécution qui est celle du noyau Linux et qui présente une trop grande complexité. Le test, en plus de pouvoir s'adapter même à un code aussi difficile à valider que celui d'un système d'information, a également l'avantage de pouvoir évoluer de manière fluide tout au long de la vie d'un système d'exploitation. En effet, du fait du nouveau matériel supporté, de fonctionnalités nouvelles, etc. le code d'un système d'exploitation est amené à être modifié très souvent. Pour rester cohérentes, les validations formelles doivent donc être menées parallèlement au cycle de développement du système. Des jeux de tests unitaires, se concentrant sur une seule portion du code à la fois permettent cela. Pour ces raisons, le test nous a paru l'approche la plus réaliste et la plus prometteuse. Comme la confiance que l'on peut au final accorder au moniteur de flux testé dépend grandement de la suite de tests, une construction systématique et formelle de celle-ci donnera des garanties suffisamment forte.

Comme nous l'avons vu plus haut, nous nous proposons de construire une sémantique des appels système en termes de flux d'information. À notre connaissance, aucun travail n'a encore été

effectué dans ce domaine. Il existe cependant des sémantiques du langage C, dans lequel sont codés la majorité des systèmes d'exploitation à l'heure actuelle [16, 1, 17]. Cependant, nous cherchons à bâtir une sémantique d'assez haut niveau, de celui du système d'exploitation, et ces sémantiques du langage C ne sont en fait pas directement exploitables car, trop fines, elles ne permettent pas de construire un modèle utilisable. Nous nous sommes donc tournés vers des langages de descriptions des processus. Le π-calcul créé par Robin Milner [14, 18] est utile pour décrire les actions de processus et leurs entrelacements. Cependant, ce formalisme ne nous était pas utile. Il s'attache trop à la communication entre processus mais ne donne pas de mécanisme pour décrire des effets de bord, pourtant nombreux dans un noyau de système d'exploitation — qu'il s'agisse de création de processus, d'allocation de mémoire, etc. Enfin, nous nous sommes intéressés à la logique transactionnelle [2, 3, 4] qui nous a donné à la fois un langage de description et un langage d'exécution, permettant l'interprétation des spécifications d'appels système pour les jeux de tests.

1.5 Plan

La figure 1 présente la démarche que nous avons suivie durant ce stage. Les bulles noires représentent les étapes que nous avions à réaliser et que j'ai mené à bien. Les bulles grises représentent les étapes qui étaient en-dehors du sujet de ce stage et qui seraient nécessaires pour avoir une solution complète intégrant Blare. Il faut noter que nous n'avions pas comme objectif premier d'arriver à une solution intégrale à la fin de ce stage mais plutôt de montrer le bien-fondé de notre approche. Pour cette raison par exemple, nous n'avons pas travaillé sur les appels système de Linux — pour lequel est implémenté `kblare` — mais Minix, dont le code plus simple et petit nous permettait de progresser plus vite.

Le reste de l'article est divisé comme suit. Dans la partie 2, nous décrivons la logique transactionnelle concurrentielle, le formalisme que nous avons choisi pour exprimer la sémantique des appels système en termes de flux d'information. Nous décrivons les principes de la logique et les raisons pour lesquels nous l'avons choisie. Dans la partie 3, nous décrivons comment nous avons étendu et adapté la logique à notre usage et comment se présente notre sémantique. Ensuite, nous présentons la prototype d'interpréteur de logique transactionnelle que nous avons construit pour l'exécution de nos jeux de tests. Dans la dernière partie, nous détaillons les résultats, y compris en termes de performances, que nous avons obtenus. Nous concluons avec les prochains travaux que nous envisageons pour améliorer notre prototype ainsi que de futures directions de recherche pour la construction de notre sémantique.

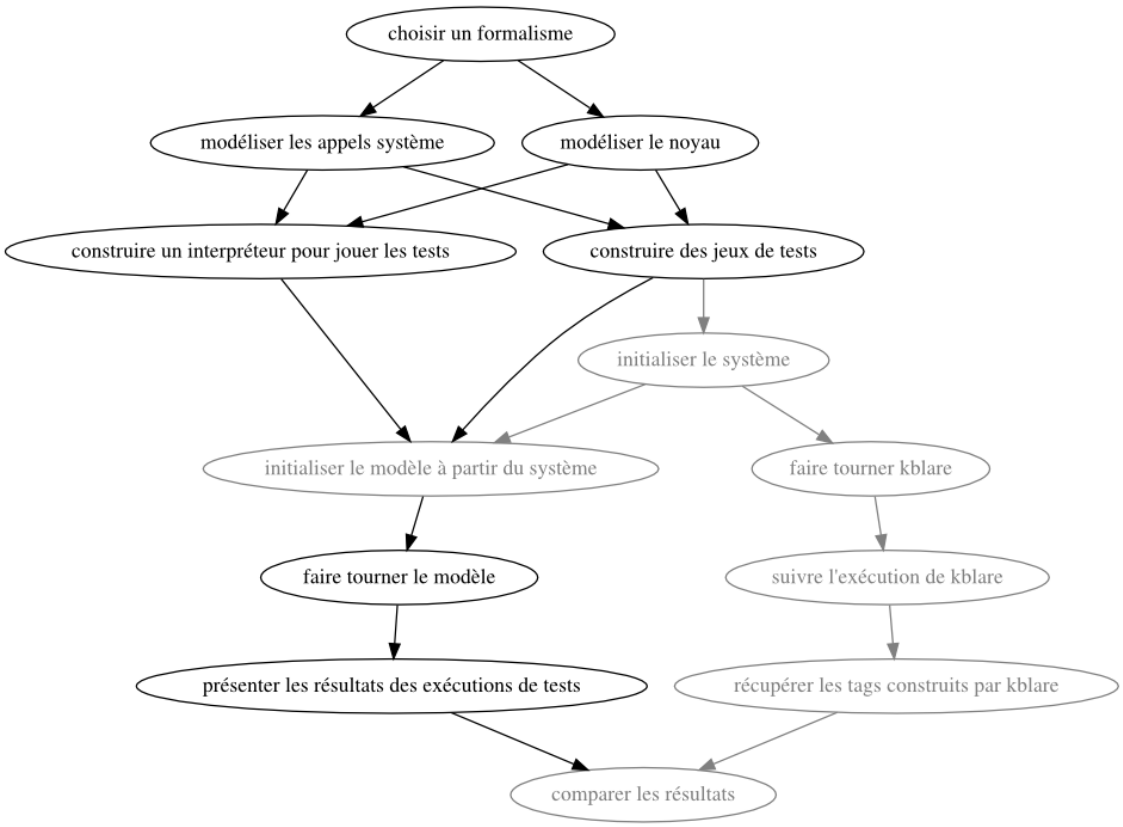

FIGURE 1 – Vue d'ensemble de notre approche

2 Logique transactionnelle concurrentielle

La logique transactionnelle concurrentielles — abrégée $\mathcal{CTR}$ d'après son nom en anglais *Concurrent TRansaction logic* — a été définie par Anthony J. Bonner et Michael Kifer [4]. La $\mathcal{CTR}$ est une extension de la logique classique du premier ordre conçue pour modéliser des *transactions*, c'est-à-dire des requêtes et des mises à jour dans une base de données. Les transactions doivent ici être comprises intuitivement comme des suites d'actions qui doivent toutes être effectuées, dans un ordre déterminé. Si la transaction ne peut pas être menée à terme — parce qu'elle entraînerait une incohérence du système par exemple — toutes les actions qu'elle comprend et qui ont déjà été effectuées doivent être annulées, et l'état de la base de données doit redevenir l'état antérieur au début de la transaction.

2.1 Syntaxe de la logique transactionnelle concurrentielle

La $\mathcal{CTR}$ est à la fois un langage déclaratif, pouvant servir à décrire les appels système, et un langage de programmation, dans lequel on peut de fait *simuler l'exécution* d'un appel système. Ceci est permis grâce aux extensions apportées par la $\mathcal{CTR}$ à la logique classique. Ces extensions consistent en deux opérateurs et une modalité. Le premier opérateur est la conjonction séquentielle, dénotée $\otimes$. Une transaction peut être composée de deux sous-transactions séparées par cet opérateur. Dans ce cas, son exécution correspond à l'exécution de la première sous-transaction *suivi de* celle

de la deuxième. Le second est la conjonction concurrentielle, dénotée |. Une transaction peut être composée de deux sous-transactions séparées par cet opérateur. Son exécution correspond alors à l'entrelacement des exécutions des deux sous-transactions : le début d'une des deux va s'exécuter, puis le début de l'autre, puis la première va s'exécuter encore en partie, puis l'autre, etc. jusqu'à ce que les deux soient entièrement terminées. L'exécution n'est donc pas déterministe, il existe plusieurs manières d'exécuter une transaction si elle comprend une conjonction concurrentielle. Cet opérateur est un ajout de Bonner et Kifer à la logique transactionnelle de base, qui constitue un de leurs travaux antérieur à la $\mathcal{CTR}$ [2, 3]. Cette logique n'avait pas de notion de concurrence ni de communication entre les processus. Enfin, de paire avec la concurrence, la $\mathcal{CTR}$ a introduit la notion d'atomicité à l'aide de la modalité d'isolation, dénotée $\odot$. La modalité d'isolation « interrompt » le fonctionnement concurrentiel normal de la logique pour passer dans un mode où seule l'exécution explicitement séquentielle existe.

2.1.1 Transactions

La $\mathcal{CTR}$ définit la notion de transaction ainsi que leur exécution. Une transaction est identique à un prédicat de la logique du premier ordre à ceci près qu'elle peut comprendre les nouveaux opérateurs et la nouvelle modalité. Dans un premier temps, nous définissons les transactions. Définir *l'exécution d'une transaction* revient en fait à décrire la sémantique de la $\mathcal{CTR}$, comme on le verra plus loin.

Une transaction est définie récursivement de la manière suivante.

Définition 1.

- () est la transaction vide.
- Un prédicat de la logique classique du premier ordre utilisant les opérateurs classiques de négation et de conjonction ($\neg$, $\wedge$) ainsi que tous ceux qui peuvent être définis à partir d'eux (comme la disjonction $\vee$ ou l'implication $\rightarrow$) est une transaction.
- Avec t_1 et t_2 deux transactions, $t_1 \otimes t_2$ est une transaction. C'est la transaction dont l'exécution correspond à celle de t_1 suivie de celle de t_2.
- Avec t_1 et t_2 deux transactions, $t_1 \mid t_2$ est une transaction. C'est la transaction dont l'exécution correspond à celle de t_1 parallèlement à celle de t_2.
- Avec t_1 une transaction, $\odot t_1$ est une transaction. Son exécution correspond à celle de t_1 effectuée de manière *isolée* ou *sans interruption*, c'est-à-dire sans qu'une autre transaction concurrente puisse voir son exécution entrelacée avec t_1.

2.1.2 Oracles

Pour écrire les transactions, il est nécessaire de pouvoir écrire des prédicats correspondants aux requêtes et aux modifications de la base de données. Le cas le plus simple pour l'écriture des transactions est celui où la base de données est une collections de prédicats, définissant des relations entre les éléments d'un ensemble de symboles. On a alors une base relationnelle, comme la base de faits utilisée par Prolog par exemple. Cependant, cela n'est pas adapté à tous les usages et Bonner et Kifer ont donc défini un mécanisme permettant l'utilisation de base de données quelconques et non uniquement relationnelles.

La $\mathcal{CTR}$ nécessite la définition de deux oracles : un oracle de données et un oracle de transitions élémentaires.

Oracle de données L'oracle de données donne pour chaque état de la base de données une représentation logique de cet état pour le « consulter ». Formellement, D étant un état de la base de données, $\mathcal{O}^d$ est l'oracle de données et $\mathcal{O}^d D$ est l'ensemble de toutes les propositions de la logique classique du premier ordre vraies pour l'état D. Bien entendu, la définition de ce qu'est un *état de la base de données* dépend de la base en question, de la nature du problème traité, etc. Comme mentionné plus haut, cette base de données peut ne pas être relationnelle. Par conséquent, la définition de l'oracle lui-même dépend de l'usage fait de la $\mathcal{CTR}$.

Oracle de transitions élémentaires De la même manière que l'oracle de données permet la consultation de la base de données, il faut un mécanisme pour permettre l'évolution de cette base à travers des mises à jour. Pour cela, un oracle de transitions élémentaires donne les transitions élémentaires correspondant chacune à un unique changement indivisible dans la base de données. Avec D_1 et D_2 deux états de la base de données et $\mathcal{O}^t$ l'oracle de transitions élémentaires, $\mathcal{O}^t D_1, D_2$ est l'ensemble de tous les prédicats de la logique classique du premier ordre qui font que la base de données passe de l'état D_1 à l'état D_2. La définition formelle requiert la définition exacte de la structure de la base de données et son fonctionnement.

Discussion Comme on le voit, les oracles constituent un paramètre de la $\mathcal{CTR}$ et permettent de définir la base de données en-dehors de celle-ci, dans un formalisme qui peut lui-même ne pas être la logique, tant que sont fournis les oracles adéquats permettant la manipulation de la mémoire comme un objet logique. On notera que la sortie de l'oracle de données est un ensemble de propositions et que celle de l'oracle de transitions élémentaires un ensemble de prédicats. En réalité, dans les deux cas, la sortie pourrait être un ensemble de prédicats, ou même de transactions, mais comme montré par Bonner et Kifer, cela n'augmenterait pas le pouvoir d'expression ou de calcul de la $\mathcal{CTR}$ [4] ; il est donc plus simple de s'en tenir à des objets logiques plus élémentaires.

Les auteurs se concentrent principalement sur les bases de données relationnelles. L'oracle de données est trivial dans ce cas car la base étant déjà relationnelle, l'oracle est simplement l'identité : il retourne tous les prédicats de la base. Le cas des transitions est plus subtil. Imaginons qu'une base de données contienne les prédicats suivants :

— `fonctionne(machine).`
— `en_panne(X) :- not(fonctionne(X)).`

La sémantique de cette base est intuitive. Informellement, on comprend que la `machine` fonctionne, et que quelque chose est en panne lorsqu'il ne fonctionne pas. À présent, imaginons que la base de données soit *dynamique*, on peut en enlever et y ajouter des prédicats. On pourrait ajouter par exemple le prédicat `not(en_panne(machine))`. Ce prédicat vient s'ajouter aux deux autres et la base reste cohérente. Mais que devient-elle si l'on ajoute aux deux prédicats ci-dessus le prédicat `en_panne(machine)` ? Que devient la valeur de vérité des deux autres ? On peut supprimer n'importe lequel des trois prédicats pour retrouver un état cohérent (au sens logique). Comment choisir entre

1. refuser la modification de la base et conserver les deux prédicats antérieurs ;
2. transformer la base de la manière suivante :
   — `fonctionne(machine).`
   — `en_panne(machine).`
   ;
3. ou de cette manière-ci :
   — `en_panne(X) :- not(fonctionne(X)).`

— `en_panne(machine).`

Intuitivement, la dernière solution semble la seule acceptable. Cependant, d'un point de vue *logique*, le problème de décision est bien plus délicat ; et bien évidemment, plus la base comporte de prédicats, plus il est difficile de ne pas introduire d'incohérences en la modifiant.

Prolog, qui n'utilise pas d'oracles mais permet la modification directe de la base de données — via les prédicats `assert/1` et `retract/2` permettant respectivement d'ajouter et de retirer de la base de données des prédicats — résout ce problème en restreignant le contenu de la base de données à des clauses de Horn [6] et en donnant une sémantique non logique à la négation. Un but de la forme `not(P)` est résolu en tentant de résoudre `P`. Si c'est impossible alors le but `not(P)` est résolu ; ce mécanisme est appelé *négation par l'échec*. Si l'évaluation de `P` ne termine pas, alors bien évidemment, celle de `not(P)` non plus. Dans ces conditions, comme il est impossible d'insérer véritablement la négation d'un fait dans la base, on ne peut pas insérer de contradiction. Cependant, cela est trop restrictif pour certaines applications, et parfois même cela ne donne pas le comportement attendu intuitivement. Typiquement, dans le cas précédent, une session Prolog serait :

```
?- ['base_initiale'] %on charge la base initiale de Prolog en mémoire
true.

?- assert(en_panne(machine)). %on insère le prédicat dans la base
true.

?- en_panne(machine).
%on demande si 'en_panne(machine)' est déductible à partir du contenu de la base
true.

?- en_panne(X).
% on demande si 'en_panne(X)' est déductible à partir du contenu
% de la base pour un certain X
X = machine.

?- fonctionne(X).
% on demande si 'fonctionne(X)' est déductible à partir du contenu
% de la base pour un certain X
X = machine.

?- not(fonctionne(X)).
% on demande si 'fonctionne(X)' n'est PAS déductible à partir du contenu
% de la base pour tout X
false.

?- not(en_panne(X)).
% on demande si 'en_panne(X)' n'est PAS déductible à partir du contenu
% de la base pour tout X
false.

?- clause(en_panne(X),A). %on vérifie qu'elles sont les clauses connues par
```

```
% Prolog pour résoudre le but "en_panne(X)"
A = not(fonctionne(X)) ; % donc Prolog connaît 'en_panne(X) :- not(fonctionne(X).'
X = machine, A = true. % et Prolog connaît 'en_panne(machine) :- true.'
```

Prolog a donc conservé tous les prédicats, même contradictoires, et retourne des résultats qui dépendent de leur ordre d'évaluation.

Dans la $\mathcal{CTR}$, les oracles résolvent ce problème en créant une indirection dans le processus de transition de la base de données. Un prédicat de transition unitaire retourné par l'oracle de transitions peut en fait correspondre à une modification arbitraire de la base de données. Pour le formuler autrement, l'expression des transitions se fait de manière logique, en revanche, les transformations induites par ces transitions ne font pas partie de la logique.

2.2 Sémantique de la logique transactionnelle concurrentielle

Comme la logique transactionnelle simple (non concurrentielle) a été conçue pour modéliser l'évolution d'une base de données, sa sémantique a été définie à travers des chemins d'exécution, qui sont des séquences d'états de la base de données [4]. Dans le cas de cette logique, on peut définir un chemin unique d'exécution, il existe un ordre total des modifications et des consultations de la base de données. En revanche, dans le cas de la logique transactionnelle concurrentielle, du fait justement de cette concurrence dans l'exécution des modifications de la base de données, il existe seulement un ordre partiel des états successifs de la base de données. La sémantique ne repose donc pas sur des chemins mais des multi-chemins.

La $\mathcal{CTR}$ a été prouvée complète et correcte pour une telle sémantique [4]. Cependant, la théorie entière servant à construire la valeur de vérité d'une transaction est très riche et nous ne la détaillerons pas ici. Nous nous intéressons plutôt à la sémantique *exécutable* de la $\mathcal{CTR}$. En effet, la $\mathcal{CTR}$ dispose d'un système d'inférence, défini dans le même article, permettant de déterminer à partir d'une transaction, du langage — c'est-à-dire des symboles de variables, de fonctions, de prédicats —, des oracles de données et de transitions, d'un état de départ de la base de données et d'un ensemble de transactions prédéfinies les séquences d'états correspondants à l'exécution de la transaction. Ce système d'inférence, nommé $\mathcal{F}_C$, est équivalent au modèle théorique de la $\mathcal{CTR}$. Il est donc complet et correct.

2.2.1 Transaction en tant que but à exécuter

Pour comprendre le système d'inférence, il est nécessaire de saisir les rapprochements entre celui-ci et les fondements de la programmation logique. Lorsque l'on s'interroge sur la valeur de vérité d'une transaction, on ne s'intéresse pas à la valeur de vérité d'une proposition dans un état comme dans la logique classique mais sur le multi-chemin, les séquences d'états, décrivant l'évolution de la base de données correspondant à la transaction. La valeur de vérité des transactions est donc définie sur ces multi-chemins et non sur des états de la base de données eux-mêmes. De plus, plus que cette valeur de vérité elle-même, on cherche en général à obtenir l'unification des variables de la transaction. On se rappelle que la $\mathcal{CTR}$ est paramétrée par un langage, comme toutes les logiques, et par deux oracles. Le langage est constituée d'un ensemble de symboles de constantes, d'un ensemble de symboles de prédicats, d'un ensemble de symboles de fonctions et enfin d'un ensemble de symboles de variables. Tous ces symboles sont des termes au sens de Herbrand. Lorsque l'on recherche un multi-chemin correspondant à la transaction à partir d'un état de départ, on obtient également une unification des variables avec des termes composés des symboles du langages. On saisit donc

intuitivement que, tout comme en programmation logique, on cherche en fait à résoudre un but logique lorsque l'on soumet une transaction à un interpréteur de la $\mathcal{CTR}$.

2.2.2 Système d'inférence

Le système d'inférence développé comprend un axiome et quatre règles d'inférence. $\mathbf{P}$ étant une base de transitions et $\mathbf{D}$, des états de la base de données. La base de transitions est un ensemble de prédicats prédéfinies. *A minima*, il s'agit des prédicats fournis par les oracles mais il peut y en avoir d'autres, construits à partir de ceux-ci. Le but du système d'inférence est de démontrer $\mathbf{P}, \mathbf{D} - - \vdash \psi$ avec ψ une transaction donnée. Informellement, cela signifie que la transaction ψ est exécutable depuis l'état de départ $\mathbf{D}$, avec les transitions $\mathbf{P}$, sans que l'on connaisse par avance les états intermédiaires ni même l'état final du système. En fait, le système d'inférence permet de *construire* ces états, lors de l'exécution de la transaction.

Définition 2 (Composantes immédiatement exécutable)**.** Les composantes immédiatement exécutables d'une transaction ϕ, dénotées $hot(\phi)$ sont définies inductivement de la manière suivante :

- $hot(()) = \emptyset$;
- $hot(b) = b$ si b est une transaction atomique (c'est-à-dire une transaction ne comprenant aucun des opérateurs de conjonction sérielle ou concurrentielle) ;
- $hot(\phi_1 \otimes \ldots \otimes \phi_n) = hot(\phi_1)$;
- $hot(\phi_1 \mid \ldots \mid \phi_n) = hot(\phi_1) \cup \cdots \cup hot(\phi_n)$;
- $hot(\odot \phi_i) = \odot \phi_i$.

Les composantes immédiatement exécutables d'une transaction correspondent intuitivement aux sous-parties de celles-ci par lesquels il est possible d'entamer l'exécution : dans une exécution séquentielle, la première, dans une exécution concurrente, n'importe laquelle des sous-parties en concurrence, etc.

Définition 3 (Axiome 0)**.**

$$\forall \mathbf{D}, \mathbf{P}, \mathbf{D} \vdash ()$$

Cet axiome signifie que la transaction vide () qui est définie comme la transaction ne faisant rien réussit toujours, et que $\mathbf{D}$ est l'état final de l'exécution de la transaction.

Définition 4 (Règle 1 *Application de la définition de la transaction*)**.** Soit $b \leftarrow \beta$ une règle de $\mathbf{P}$. Si a et b s'unifient par l'unifieur le plus général σ, et en renommant si besoin les variables de cette règle pour qu'aucune n'apparaissent dans ψ,

$$\frac{\mathbf{P}, \mathbf{D} - - \vdash \psi'\sigma}{\mathbf{P}, \mathbf{D} - - \vdash \psi}$$

où ψ' est obtenu à partir de ψ en remplaçant la composante $a \in hot(\phi)$ par β

Cette règle dit que si la première étape des étapes restantes d'une transaction est le nom d'une règle (une transaction connue), on peut remplacer ce nom par le corps de cette règle, tant que l'unification est possible. L'état de la base ne change pas.

Définition 5 (Règle 2 *Requête de la base de données*)**.** Si $\mathcal{O}^d(D) \models a\sigma$, et que $a\sigma$ et $\psi'\sigma$ ne partagent aucune variable

$$\frac{\mathbf{P}, \mathbf{D} - - \vdash \psi'\sigma}{\mathbf{P}, \mathbf{D} - - \vdash \psi}$$

où ψ' est obtenu à partir de ψ en supprimant la composante $a \in hot(\psi)$

Cette règle dit que l'on peut exécuter la première étape des étapes restantes de la transaction sans changer l'état de la base de données si l'étape à exécuter correspond à un prédicat vrai dans l'état courant de la base de données. En d'autres termes, la consultation de la base de données simplifie le but à résoudre — il y a une étape de moins — sans faire évoluer la base.

Définition 6 (Règle 3 *Mise à jour élémentaire de la base de données*)**.** Si $\mathcal{O}^t(D_1, D_2) \models a\sigma$, et que $a\sigma$ et $\psi'\sigma$ ne partagent aucune variable

$$\frac{\mathbf{P}, \mathbf{D}_2 - \ - \vdash \psi'\sigma}{\mathbf{P}, \mathbf{D}_1 - \ - \vdash \psi}$$

où ψ' est obtenu à partir de ψ en supprimant la composante $a \in \psi$

Cette règle dit que l'on peut exécuter la première étape des étapes restantes de la transaction correspond à un prédicat de la base de transitions élémentaires alors on peut exécuter cette transition, ce qui entraîne un changement unitaire de la base de données. Le but à résoudre avance d'une étape.

Définition 7 (Règle 4 *Exécution atomique*)**.** Si $\odot a$ est une composante immédiatement exécutable de ψ

$$\frac{\mathbf{P}, \mathbf{D} - \ - \vdash (a \otimes \psi')}{\mathbf{P}, \mathbf{D} - \ - \vdash \psi}$$

où ψ' est obtenu à partir de ψ en supprimant la composante $a \in \psi$.

Cette règle dit que si la transaction comporte comme étape immédiatement exécutable une série d'actions à effectuer de manière atomique, alors on peut extraire cette composante, l'exécuter intégralement en premier, puis exécuter le reste de la transaction ensuite.

Ces définitions mettent en avant, à travers la notion de composante immédiatement exécutable, le non-déterminisme de l'interpréteur. N'importe quelle composante immédiatement exécutable peut être choisie pour être exécutée en premier.

2.3 Limitations de la $\mathcal{CTR}$

Bien que la $\mathcal{CTR}$ soit un outil très performant et très puissant pour décrire nos appels système, comme on le verra dans la prochaine section, elle souffre de limitations sérieuses qui restreignent son domaine d'application. Premièrement, elle ne permet pas de décrire des modalités temporelles. Par conséquent, nous n'avons pas d'horloge dans notre modèle de système d'exploitation. Ceci nous empêche donc de modéliser certains appels système basés sur le temps, comme `sleep` permettant à un processus de s'endormir pendant un certain temps ou `alarm` qui permet à un processus de demander l'envoi d'un signal `ALARM` après une durée spécifiée. Ceci a une conséquence directe sur la correction de notre modèle puisque nous sommes donc incapables de traiter les flux d'information sur les canaux cachés temporels. Deuxièmement, nous ne pouvons pas gérer les évènements asynchrones dans notre système. En particulier, l'envoi et la réception de signaux ne peuvent pas être modélisés dans le cas général. Ceci est une limitation certaine car les signaux constituent de fait un flux d'information entre processus. Le simple numéro de signal envoyé véhicule plusieurs bits d'information. Néanmoins, le cas où les signaux sont utilisés de manière synchrone — les processus se bloquent pour *attendre* les signaux émis par d'autres processus via un appel à `pause` par exemple — peut lui être modélisé car il s'agit alors d'une synchronisation par rendez-vous, or la $\mathcal{CTR}$ a été conçue modéliser ce type de communication [4].

Nom	Structure	Fonction
`k_procs`	Table des processus	Mémorise pour chaque processus ses méta-données
`k_fdesc`	Table des descripteurs de fichiers	Mémorise pour chaque fichier ouvert quelques informations
`k_fs`	Table du système de fichiers	Mémorise pour chaque fichier existant dans le système ses données et méta-données
`k_mem`	Table de la mémoire	Mémorise pour chaque bloc de mémoire alloué ses données et le fait d'être partageable ou non

Table 1 – Structures de données du noyau

3 Descriptions des appels système

Dans notre cas, la base de données représente les informations connues du noyau à propos de l'état de système. Nous avons choisi cette représentation pour nous placer du point de vue du noyau. En effet, `kblare` est implémenté comme un *patch* dans le noyau Linux et a donc une vue limitée de l'activité du système d'exploitation. En particulier, toutes les actions menées par les processus en espace utilisateur ne sont pas vues par `kblare`. En revanche, le noyau a une connaissance parfaite de l'usage des ressources par les processus et des appels système utilisés. Dans notre modélisation, les informations connues du noyau concernent les processus, les fichiers, les descripteurs de fichiers utilisés, la mémoire, et les flux d'information enregistrés jusqu'alors entre ceux-ci. Cette modélisation est la première contribution de ce stage.

3.1 Structures du noyau

La base de données est constituée de quatre tables : `k_procs` pour les processus, `k_fs` pour les fichiers, `k_fdesc` pour les descripteurs de fichiers et `k_mem` pour la mémoire ainsi que d'une liste de flux `k_flows`. Les tables associent des index uniques à des entrées, toutes de format identique. Les entrées comportent un nombre déterminé de champs, chacun comprenant une valeur d'un type prédéfini. La liste de flux est une liste totalement ordonnée d'entrées correspondant à des flux. À la différence de la base de données manipulée par Prolog (qui est une liste de faits et de clauses), son schéma est fixe et elle n'est pas déductive car elle ne comporte pas de clauses mais uniquement des faits. La table 1 présente les structures de données du noyau.

3.1.1 Détails des champs des structures

`k_procs` La table `k_procs` stocke les processus actifs dans le système. C'est la table comportant le plus de champs dans notre représentation. Ses champs sont détaillés dans la table 2.

`k_fdesc` Dans la table `k_fdesc`, nous stockons les descripteurs de fichiers ouverts dans le système. Contrairement à un véritable système UNIX où chaque processus dispose d'une table de descripteurs de fichiers séparée, dans notre modèle, nous stockons tous les descripteurs de fichiers de tous les processus dans la même table. Cela facilite la modélisation des descripteurs de fichiers partagés entre processus. Cependant, cela nécessite de maintenir d'une autre manière la vue qu'ont les processus de leur descripteur. Cela est réalisé grâce au champ `fd` de la table qui contient le descripteur de fichier

Nom court	Nom développé	Type	Signification
pid	Process IDentifier	Entier	identifiant unique du processus
ppid	Parent Process IDentifier	Entier	identifiant du père du processus
grpid	GRoup IDentifier	Entier	identifiant unique du groupe de processus
uid	User IDentifier	Entier	identifiant unique de l'utilisateur propriétaire du processus
euid	Effective User IDentifier	Entier	identifiant unique de l'utilisateur avec les droits duquel s'exécute le processus
gid	Group IDentifier	Entier	identifiant unique du groupe de l'utilisateur propriétaire du processus
egid	Effective Group IDentifier	Entier	identifiant unique du groupe avec les droits duquel s'exécute le processus
sgrps	Supplementary GRouPS	Ensemble d'entiers	Autres groupes auxquels appartient le processus
umask	User file creation mode MASK	Entier	permissions maximum sur les nouveaux fichiers créés
root	Root directory	Fichier	Racine de l'arborescence du système de fichiers tel que vue par le processus
wd	Working Directory	Fichier	Répertoire courant du processus (à partir duquel les chemins relatifs sont interprétés)
text_mem	TEXT area MEMory	Bloc mémoire	bloc de la mémoire contenant le programme exécuté par le processus et ses données statiques
stack_mem	STACK area MEMory	Bloc mémoire	bloc de la mémoire contenant la pile d'exécution du processus
data_mem	DATA area MEMory	Bloc mémoire	bloc de la mémoire contenant les données dynamiques du processus
shared_mem	SHARED area MEMory	Ensemble de blocs mémoire	blocs de la mémoire partagées avec d'autres processus
status	Process status	Valeur d'énumération	statut du processus parmi les valeurs suivantes : $RUNNING$, $READY$, $BLOCKED\_XXX$, $DEFUNCT$
end	Ending status	Entier	valeur de retour du programme
name	Program Name	Chaîne de caractères	nom du programme exécuté

TABLE 2 – Données associées à un processus

Nom court	Nom développé	Type	Signification
file		Entrée de `k_fs`	fichier correspondant au descripteur de fichier
procs	Processus	Ensemble d'entiers	liste des PID des processus possédant un accès à ce descripteur de fichier
opts	OPTionS	Ensemble de valeurs d'énumération	sous-ensemble des valeurs possibles suivantes : $\{READ, WRITE, APPEND\}$
fd	File Descriptor	Entier	descripteur de fichier proprement dit, entier que les processus doivent donner à chaque appel système concernant le fichier ouvert
pos	POSition	Entier	position exprimée en décalage depuis le début du fichier des prochains octets lus ou écrits

TABLE 3 – Information rattaché à un descripteur de fichier

au sens UNIX, c'est-à-dire l'entier utilisable par les processus comme argument d'appels système. En quelque sorte, on donne à chaque processus une vue différente de cette table que seul le noyau connaît en entier. Voir le tableau 3 pour le détail de ses champs.

`k_fs` Cette table stocke les informations relatives aux fichiers existants dans le système. Cette vue est très primitive et réductrice, en comparaison d'un véritable système UNIX. En particulier, elle confond le concept d'*inode* (conteneur d'information à proprement parler) et d'entrée dans le système de fichiers (nom et position dans l'arborescence des fichiers). Par exemple, nous ne considérons qu'un seul chemin d'accès pour chaque fichier, qui est celui obtenu en remontant directement jusqu'à la racine du système de fichier et en imprimant les dossiers traversés. Nous nous sommes intéressés dans un premier temps aux fichiers réguliers uniquement. Le tableau 4 présente les différents champs de cette table.

`k_mem` La représentation de la mémoire est celle qui s'éloigne le plus de la véritable implémentation dans les systèmes UNIX. La modélisation de la mémoire dans notre cas est peu orthodoxe. Pour éviter d'inutiles complications, nous avons décidé de nous séparer radicalement de l'implémentation et en particulier de ne pas évoquer les concepts d'adressage virtuel, physique, de *Memory Management Unit*, etc. Dans son rôle de « distributeur de mémoire », le noyau est vu comme une boîte noire. Cette table n'a que deux champs, présentés dans la table 5.

Le noyau assure les fonctions suivantes :

— Sur demande, dans la limite des permissions des processus, il distribue à ces derniers des blocs de mémoire de taille demandée. Ces blocs de mémoire peuvent être vus comme une suite contigüe d'octets. Ils sont de taille variable. La quantité totale de mémoire distribuable peut être contrôlée par le paramètre `MAX_MEMORY` de la modélisation. Cependant, ce paramètre n'est destiné à servir que dans les cas « aux limites » où l'on teste justement le comportement du système en cas de défaut de mémoire disponible.

— Sur demande, en fonction des permissions, il peut créer un bloc partageable. Cela permet ensuite à un processus de demander ce bloc en particulier. Si plusieurs processus demandent

Nom court	Nom développé	Type	Signification
type		Valeur d'énumération	f fichier régulier d répertoire p tuyau c fichier spécial en mode caractère b fichier spécial en mode bloc l lien symbolique
path		Chaîne de caractères	chemin d'accès de l'entrée du système de fichier
rd_locked	ReaD LOCKED	Bit	indique si le fichier est verrouillé en lecture
wr_locked	WRite LOCKED	Bit	indique si le fichier est verrouillé en écriture
uid	User IDentifier	Entier	identifiant du propriétaire du fichier
gid	Group IDentifier	Entier	identifiant du groupe propriétaire du fichier
mode		Quatre chiffres octaux	droits d'accès au fichier et flags `setuid`, `setgid`, `sticky bit`

TABLE 4 – Informations rattachées à un fichier

Nom court	Nom développé	Type	Signification
shblock	SHared BLOCK	Bit	indique si le bloc est partageable
size		Entier	taille du bloc en octets

TABLE 5 – Gestion de la mémoire

le même bloc partageable, ils disposent alors d'un espace de mémoire partagée pour pouvoir s'échanger des données.

Notre représentation est une simplification assez importante par rapport aux structures de données du noyau mais elle permet de représenter malgré tout l'intégralité des conteneurs d'information du système. Nous avons éliminé les données redondantes et les mécanismes de verrous ou d'exclusion mutuels qui ne sont pas pertinentes pour notre modélisation puisque nous n'avons pas le même souci que le véritable système de maîtriser l'accès aux structures de données puisque l'exécution des processus est une abstraction dans notre modèle. Pour la même raison, nous n'avons pas besoin de modéliser la pile du noyau ou les registres de sauvegarde des processus utilisés par le système en cas de changement de contexte.

3.1.2 Opérations sur les structures de données

Pour faciliter l'écriture des appels système dans le formalisme que nous avons choisi, nous avons augmenté la syntaxe de la $\mathcal{CTR}$ avec les extensions présentées dans les tables 6 et 7. Les opérations présentées dans cette table correspondent en fait à l'interface exposée respectivement par l'oracle de données et par l'oracle de transitions élémentaires.

Nom	Syntaxe	Précautions et effet
récupération d'une entrée	*table*[*index*]	*table* doit être `k_procs`, `k_fdesc`, `k_fs`, `k_mem` *index* doit être du type de la variable d'indexation de *table* renvoit l'entrée de *table* d'index *index*
récupération d'un champ	*elem.champ*	*elem* doit être l'entrée d'une table où *champ* est disponible renvoit la valeur du champ *champ* de l'élément *elem*

TABLE 6 – Opérations de consultation sur les structures de données du noyau

Nom	Syntaxe	Précautions et effet
mise à jour d'un champ	*elem.champ* := *valeur*	*elem* doit être l'entrée d'une table où *champ* est disponible et *val* doit être du type correct met la valeur du champ *champ* de *elem* à la valeur *valeur*
suppression d'une entrée	*table.delete*(*index*)	*table* doit être `k_procs`, `k_fdesc`, `k_fs`, `k_mem` *index* doit être un index de *table* supprime l'entrée d'index *index* de *table*
création d'une entrée	*table.new*(*index*)	*table* doit être `k_procs`, `k_fdesc`, `k_fs`, `k_mem` crée une entrée dans *table* et unifie *index* avec son index
enregistrement d'un flux	*obj*1 *FLUX obj*2	*FLUX* étant un opérateur de flux (<,>,≪,≫), ajoute le flux correspondant à la fin de `k_flows`. *obj*1 et *obj*2 doivent être des entrées de `k_mem` ou `k_fs`

TABLE 7 – Opérations de modification sur les structures de données du noyau

3.2 Définition des oracles de données et de transition

3.2.1 Oracle de données

Afin de définir notre oracle de données — pour rappel, il s'agit de l'oracle nous fournissant pour chaque état de la base de données, l'ensemble des prédicats logiques du premier ordre vrais dans cet état — nous nous munissons des prédicats suivants.

— p_valueInTable(Table, Index, Field, Value)
— p_flow(Index, Source, Dest)

Notre oracle de données $\mathcal{O}^d$ est défini formellement comme suit.

Définition 8 (Oracle de données)**.**

— $p\_valueInTable(Table, Index, Field, Value) \in \mathcal{O}^d(D)$ si et seulement si il existe dans l'état D une table $Table$, qu'elle a une entrée d'index $Index$, que cette entrée possède un champ $Field$, et que ce champ est de valeur $Value$.
— $p\_flow(Index, Source, Dest, Append) \in \mathcal{O}^d(D)$ si et seulement si il existe dans l'état D un élément d'indice $Index$ dans la liste des flux et qu'il correspond à un flux de $Source$ vers $Dest$, de mode $Append$. $Source$ et $Dest$ sont des indices des tables `k_fs` ou `k_mem`. $Append$ est une valeur booléene indiquant si le flux a remplacé le contenu de $Dest$ ou l'a complété.
— dans tous les autres cas, $d \notin \mathcal{O}^d$.

Syntaxe des requêtes de la base de données Nous avons adopté une syntaxe plus proche d'un langage impératif pour notre formalisme. La correspondance de nos constructions syntaxiques avec les prédicats est donnée par la définition suivante.

Définition 9. Si dans une clause, un terme Var s'unifie avec un terme de la forme `Table`$[Index].Field$, le terme Var peut être remplacée par $Value$ si et seulement si $p\_valueInTable(Table, Index, Field, Value)$.

3.2.2 Oracle de transition

Comme dans le cas de la base de données, les transitions élémentaires sont en fait accédées à travers un oracle. Les prédicats suivants sont définis.

p_change(Table,Index,Field,Value) Ce prédicat décrit le positionnement du champ `Field` de l'entrée d'index `Index` de la table `Table` à la valeur `Value`.

p_add_flow(Source,Dest,Append) Ce prédicat décrit l'enregistrement d'un flux d'information entre `Source` et `Dest`, en mode « append » ou non, selon la valeur du booléen `Append`

p_create_entry(Table,Index) Ce prédicat décrit l'ajout d'une entrée dans la table `Table` d'index `Index`.

p_delete_entry(Table,Index) Ce prédicat décrit la suppression de l'entrée de la table `Table` d'index `Index`.

Définition 10 (Oracle de transition)**.**

— $p\_change(Table, Index, Field, Value) \in \mathcal{O}^t(D_1, D_2)$ si et seulement si

$$\exists Table, Index, Field, \mathcal{O}^d(D_2) = \mathcal{O}^d(D_1) \cup p\_valueInTable(Table, Index, Field, Value)$$

$$\setminus p\_valueInTable(Table, Index, Field, OldValue)$$
$$\text{avec } p\_valueInTable(Table, Index, Field, OldValue) \in \mathcal{O}^d(D_1)$$

Ce prédicat décrit un changement *unitaire* dans la base de données. Un prédicat $p\_valueInTable$ était vrai pour l'état D_1, ne l'est plus dans l'état D_2 alors qu'un autre avec les mêmes trois premiers paramètres (table, index et champ) est devenu vrai. Aucun autre élément de la base de donnée n'a été modifié.

— $p\_add\_flow(Source, Dest, Append) \in \mathcal{O}^t(D_1, D_2)$ si et seulement si

$$\begin{aligned}
&\mathcal{O}^d(D_2) = \mathcal{O}^d(D_1) \\
&\quad \cup p_flow(1, Source, Dest, Append) \\
&\qquad \text{si } (\neg\exists Index, OtherSource, OtherDest, OtherAppend, \\
&\qquad p_flow(Index, OtherSource, OtherDest, OtherAppend) \in \mathcal{O}^d(D_1)) \\
&\quad \vee \\
&\exists Index, \mathcal{O}^d(D_2) = \mathcal{O}^d(D_1) \\
&\quad \cup p_flow(Index + 1, Source, Dest, Append) \\
&\qquad \text{si } \exists Index, OtherSource, OtherDest, OtherAppend, \\
&\qquad p_flow(Index, OtherSource, OtherDest, OtherAppend \in \mathcal{O}^d(D_1) \\
&\qquad \wedge (\neg\exists OtherIndex, OtherSource_2, OtherDest_2, OtherAppend_2, \\
&\qquad\quad p_flow(OtherIndex, OtherSource_2, OtherDest_2, OtherAppend_2) \\
&\qquad\quad \wedge OtherIndex > Index)
\end{aligned}$$

Ce prédicat ajoute un flux à la liste des flux enregistré dans le système en le numérotant. Ce qui rend l'expression de ce prédicat compliquée est le fait que le premier flux doit être numéroté « 1 » tandis que s'il existe déjà au moins un flux enregistré, tout nouveau flux reçoit pour index l'index maximal de la liste plus un.

— $p\_add\_entry(Table, Index) \in \mathcal{O}^t(D_1, D_2)$ si et seulement si

$$\begin{aligned}
&\exists Field, Value, \mathcal{O}^d(D_2) = \mathcal{O}^d(D_1) \\
&\qquad \cup p_valueInTable(Table, Index, Field, Value) \in \mathcal{O}^d(D_2) \\
&\quad \wedge\neg\exists OtherField, OtherValue, \\
&\qquad p_valueInTable(Table, Index, OtherField, OtherValue) \in \mathcal{O}^d(D_1)
\end{aligned}$$

L'ajout d'une entrée dans une table implique la vérification préalable qu'il n'existe pas d'entrée d'index identique au nouveau (du moins, avec un contenu différent). Contrairement au prédicat précédent, la façon dont le nouvel index est choisi n'est pas spécifiée.

— $p\_delete\_entry(Table, Index) \in \mathcal{O}^t(D_1, D_2)$ si et seulement si

$$\begin{aligned}
&\exists Field, Value, p_valueInTable(Table, Index, Field, Value) \in \mathcal{O}^d(D_1) \\
&\wedge \mathcal{O}^d(D_2) = \mathcal{O}^d(D_1) \\
&\quad \setminus \{p_valueInTable(Table, Index, Field, Value) | \\
&\qquad p_valueInTable(Table, Index, Field, Value) \in \mathcal{O}^d(D_1)\}
\end{aligned}$$

Ce prédicat exprime le fait que la destruction d'une entrée nécessite son existence préalable dans la table.

Syntaxe des transitions Il peut sembler que les constructions syntaxiques introduites dans la table 7 sont assez éloignées des prédicats de l'oracle de transition mais en réalité la correspondance est directe. Plus précisément,

- si Var s'unifie avec un terme de la forme `Table`$[Index].Field$ alors la construction syntaxique $Var := Value$ peut être remplacée par $p\_change(Table, Index, Field, Value)$;
- si $FLUX$ est $\ll$ alors la construction $Obj1\ FLUX\ Obj2$ peut être remplacé par $p\_add\_flow(Obj2, Obj1, true)$;
- si $FLUX$ est $<$ alors la construction $Obj1\ FLUX\ Obj2$ peut être remplacé par $p\_add\_flow(Obj2, Obj1, false)$;
- si $FLUX$ est $\gg$ alors la construction $Obj1\ FLUX\ Obj2$ peut être remplacé par $p\_add\_flow(Obj1, Obj2, true)$;
- si $FLUX$ est $>$ alors la construction $Obj1\ FLUX\ Obj2$ peut être remplacé par $p\_add\_flow(Obj1, Obj2, false)$;
- la construction $Table.new(-Index)$ peut être remplacée par $p\_create\_entry(Table, Index)$;
- la construction $Table.delete(+Index)$ peut être remplacée par $p\_delete\_entry(Table, Index)$.

3.3 Exemple d'appel système : `read`

Nous avons rédigé dans la syntaxe de notre modèle la description d'une quinzaine d'appels système. Certains sont très simples, comme `getuid` ou `getpid`, qui récupèrent seulement une information depuis les tables de la base de données. D'autres ont une sémantique nettement plus riche et complexe comme `fork` ou `open`. La liste des appels système rédigés est présentée en annexe A.1. Dans cette section, nous détaillons un appel système en particulier, `read`, comme exemple.

3.3.1 Macro

Certains appels système partagent une partie de leur code, comme par exemple pour faire la vérification d'une certaine condition du système, ou récupérer l'index d'une certaine valeur dans une table. C'est le cas de `read` avec `GET_FID_FOR_PROC` qui partage en partie son code avec l'appel système `write` qui sert à écrire dans un fichier.

$$\begin{aligned}&\texttt{GET\_FID\_FOR\_PROC}(+\text{Proc},+\text{Fd},-\text{Fid}) \leftarrow \\ &\quad (\texttt{k\_fdesc}[F].procs \ni Proc \wedge \texttt{k\_fdesc}[F].fd = Fd) \\ &\quad \otimes Fid = \texttt{k\_fdesc}[F].\end{aligned}$$

Cette macro est utile pour récupérer l'index dans la table `k_fdesc` de l'entrée correspondant à un descripteur de fichier d'un processus en particulier. Plusieurs processus peuvent avoir un descripteur de fichier 0 par exemple, mais chacun n'en n'a au plus qu'un seul. Ce descripteur représente l'entrée standard du processus. Pour faire les manipulations sur l'entrée de la table `k_fdesc`, il faut croiser l'information du descripteur de fichier et du processus pour en retrouver l'index dans la table.

Cette macro prend en paramètre un processus (un index dans la table `k_procs`, un descripteur de fichier (identifiant numérique connu du processus) et une variable non liée [2]. La clause décrit le cas où il est possible de trouver dans la table `k_fdesc` une entrée correspondant à un descripteur de fichier ouvert pour le processus en paramètre et de numéro égal au deuxième paramètre. Dans ce cas, le troisième paramètre, la variable, est unifiée à l'index de cette entrée. S'il n'existe pas de telle entrée, la macro échoue.

3.3.2 Appel système

$$
\begin{aligned}
&\texttt{read}(+\text{FileDescriptor}, +\text{Buffer}, +\text{Size}, -\text{Return}) \leftarrow \\
&\quad not(\texttt{GET\_FID\_FOR\_PROC}(caller, FileDescriptor, Fid)) \\
&\quad \otimes Return = \texttt{EBADF}. \\
&\texttt{read}(+\text{FileDescriptor}, +\text{Buffer}, +\text{Size}, -\text{Return}) \leftarrow \\
&\quad \texttt{GET\_FID\_FOR\_PROC}(caller, FileDescriptor, Fid) \\
&\quad \otimes Fid.opts \not\ni \texttt{READ} \\
&\quad \otimes Return = \texttt{EIO}. \\
&\texttt{read}(+\text{FileDescriptor}, +\text{Buffer}, +\text{Size}, -\text{Return}) \leftarrow \\
&\quad \texttt{GET\_FID\_FOR\_PROC}(caller, FileDescriptor, Fid) \\
&\quad \otimes Fid.opts \ni \texttt{READ} \\
&\quad \otimes Size = 0 \\
&\quad \otimes Return = 0. \\
&\texttt{read}(+\text{FileDescriptor}, +\text{Buffer}, +\text{Size}, -\text{Return}) \leftarrow \\
&\quad \texttt{GET\_FID\_FOR\_PROC}(caller, FileDescriptor, Fid) \\
&\quad \otimes Fid.opts \ni \texttt{READ} \\
&\quad \otimes Size > 0 \\
&\quad \otimes ReqSize \ is \ max(Size, Fid.file.size - Fid.pos) \\
&\quad \otimes Buffer \ll Fid.file \\
&\quad \otimes Fid.pos := Fid.pos + ReqSize \\
&\quad \otimes Return = ReqSize.
\end{aligned}
$$

L'appel système `read` est écrit en quatre clauses. Les deux premières décrivent des cas d'erreurs tandis que les deux dernières correspondent aux cas où l'opération se déroule correctement. `read` prend quatre arguments en paramètres : un descripteur de fichiers, une zone mémoire (un index de la table `k_mem`), une taille de buffer et une variable non liée.

La première étape de toutes les clauses est un appel à `GET_FID_FOR_PROC`, qui doit renvoyer l'index de `k_procs`. La première clause décrit le cas où cette macro échoue. Dans ce cas, `read` unifie la variable de retour (le quatrième paramètre) avec `EBADF` qui est un code d'erreur indiquant que le descripteur de fichier passé en paramètre est invalide.

2. c'est-à-dire une variable libre, qui n'a encore été unifiée à aucune valeur

Le deuxième test qui est effectué est la vérification des options du descripteurs de fichiers. Il doit avoir été ouvert en lecture. Sinon, la valeur d'erreur `EIO` est unifiée avec la variable de retour. Ensuite, la taille de la requête est lue. Si elle est de zéro, aucune manipulation n'est à faire dans les tables du noyau, et `read` unifie la variable de retour avec zéro pour indiquer que tout s'est bien passé.

La quatrième clause décrit le cas où une véritable opération de lecture est nécessaire, c'est-à-dire lorsque le processus appelant demande à lire strictement plus de zéro octets. Dans ce cas, la taille à lire est ajustée au maximum lisible (pour ne pas lire plus d'octets qu'il n'en reste à lire de la position courante jusqu'à la fin du fichier). Ensuite, un flux est enregistré du fichier vers la zone mémoire, puis la position de lecture est déplacée de la taille lue. Enfin, le nombre d'octets lus est unifié à la variable de retour.

3.4 Exemple d'utilisation du système d'inférence

Dans cette section, nous présentons l'utilisation du système d'inférence à travers un exemple. Normalement, le système d'inférence n'est pas utilisé directement pour réaliser des preuves ; il est utilisé par l'interpréteur que nous avons écrit, de manière automatisée. Cependant, il est utile pour comprendre le fonctionnement de notre modèle, de voir l'exécution du système d'inférence sur un exemple. Ici, nous prenons un cas très simple, ne faisant même pas intervenir de concurrence entre processus. Nous avons un seul processus et deux fichiers. Le processus lit tout d'abord dans le premier fichier avec l'appel système `read` puis écrit dans le deuxième avec l'appel système `write`.

3.4.1 Situation initiale

La situation de départ est présentée informellement dans la table 8.

fs	fichier 1	fichier 2
type	f	f
uid	procA	procB
gid	procA	procA
mode	`rw-r-----`	`rw-rw----`
size	128	512

(a) Système de fichiers

fdesc	1	2
file	fichier 1	fichier 2
procs	{procA}	{procA}
opts	{READ}	{APPEND}
fd	3	4
pos	0	0

(b) Descripteurs de fichiers

mem	stack	data
size	5000	10000

(c) Mémoire

proc	procA
euid	1000
egid	1000
stack_mem	stack

(d) Processus

TABLE 8 – Situation initiale

Plus formellement, l'état de départ est tel que l'oracle $\mathcal{O}^d$ donne pour l'état de départ, l'ensemble des prédicats suivants :

— $p\_valueInTable(\texttt{k\_fs}, fichier1, type, f)$;

— $p\_valueInTable(\texttt{k\_fs}, fichier1, uid, \texttt{k\_procs}[procA])$;
— $p\_valueInTable(\texttt{k\_fs}, fichier1, gid, \texttt{k\_procs}[procA])$;
— $p\_valueInTable(\texttt{k\_fs}, fichier1, mode, \texttt{rw-r----})$;
— $p\_valueInTable(\texttt{k\_fs}, fichier1, size, 128)$;

— $p\_valueInTable(\texttt{k\_fs}, fichier2, type, f)$;
— $p\_valueInTable(\texttt{k\_fs}, fichier2, uid, \texttt{k\_procs}[procB])$;
— $p\_valueInTable(\texttt{k\_fs}, fichier2, gid, \texttt{k\_procs}[procA])$;
— $p\_valueInTable(\texttt{k\_fs}, fichier2, mode, \texttt{rw-rw----})$;
— $p\_valueInTable(\texttt{k\_fs}, fichier2, size, 512)$;

— $p\_valueInTable(\texttt{k\_fdesc}, 1, file, \texttt{k\_fs}[fichier1])$;
— $p\_valueInTable(\texttt{k\_fdesc}, 1, procs, \{\texttt{k\_procs}[procA]\})$;
— $p\_valueInTable(\texttt{k\_fdesc}, 1, opts, \{READ\})$;
— $p\_valueInTable(\texttt{k\_fdesc}, 1, fd, 3)$;
— $p\_valueInTable(\texttt{k\_fdesc}, 1, pos, 0)$;

— $p\_valueInTable(\texttt{k\_fdesc}, 2, file, \texttt{k\_fs}[fichier2])$;
— $p\_valueInTable(\texttt{k\_fdesc}, 2, procs, \{\texttt{k\_procs}[procA]\})$;
— $p\_valueInTable(\texttt{k\_fdesc}, 2, opts, \{APPEND\})$;
— $p\_valueInTable(\texttt{k\_fdesc}, 2, fd, 4)$;
— $p\_valueInTable(\texttt{k\_fdesc}, 2, pos, 0)$;

— $p\_valueInTable(\texttt{k\_mem}, stack, size, 5000)$;

— $p\_valueInTable(\texttt{k\_mem}, data, size, 10000)$;

— $p\_valueInTable(\texttt{k\_procs}, procA, euid, 1000)$;
— $p\_valueInTable(\texttt{k\_procs}, procA, egid, 1000)$;
— $p\_valueInTable(\texttt{k\_procs}, procA, stack\_mem, \texttt{k\_mem}[stack])$.

On cherche à démontrer que l'exécution par le processus $\mathcal{A}$ des deux appels système ci-dessous entraîne l'apparition de deux flux : le premier du fichier1 vers le processus $\mathcal{A}$ et le second du processus $\mathcal{A}$ au fichier2.

$$read(+FileDescriptor = 3, +Buffer = \texttt{k\_mem}[stack], +Size = 128, -Return) \quad (1)$$
$$\otimes write(+FileDescriptor = 4, +Buffer = \texttt{k\_mem}[stack], +Size = 128, -Return) \quad (2)$$

Démonstration Nous allons réaliser la démonstration grâce au système d'inférence. Soit $\mathbf{D_0}$ l'état initial décrit plus haut. Le système d'inférence fait intervenir une base de transitions $\mathbf{P}$. Normalement, il s'agirait de notre ensemble d'appels systèmes et de macros définis mais pour l'exemple nous pouvons le restreindre à l'ensemble de clauses suivant :

$$\begin{aligned} &\texttt{GET\_FID\_FOR\_PROC}(+\text{Proc},+\text{Fd},-\text{Fid}) \leftarrow \\ &\quad (\texttt{k\_fdesc}[F].procs \ni caller \wedge \texttt{k\_fdesc}[F].fd = Fd) \\ &\quad \otimes Fid = \texttt{k\_fdesc}[F]. \end{aligned} \quad (3)$$

$$
\begin{aligned}
&\texttt{read}(+\text{FileDescriptor}, +\text{Buffer}, +\text{Size}, \text{-Return}) \leftarrow \\
&\quad \texttt{GET\_FID\_FOR\_PROC}(caller, FileDescriptor, Fid) \\
&\quad \otimes max(Size, Fid.file.size - Fid.pos, ReqSize) \\
&\quad \otimes Buffer \ll Fid.file \\
&\quad \otimes Fid.pos := Fid.pos + ReqSize \\
&\quad \otimes Return = ReqSize.
\end{aligned} \tag{4}
$$

$$
\begin{aligned}
&\texttt{write}(+\text{FileDescriptor}, +\text{Buffer}, +\text{Size}, \text{-Return}) \leftarrow \\
&\quad \texttt{GET\_FID\_FOR\_PROC}(caller, FileDescriptor, Fid) \\
&\quad \otimes Fid.opts \ni \texttt{APPEND} \\
&\quad \otimes Fid.pos := Fid.file.size \\
&\quad \otimes Fid.file \gg Buffer \\
&\quad \otimes Fid.pos := Fid.pos + Size \\
&\quad \otimes Fid.file.size := Fid.file.size + Size \\
&\quad \otimes Return = Size.
\end{aligned} \tag{5}
$$

On appelle ϕ la transaction $\texttt{read}(\cdots) \otimes \texttt{write}(\cdots)$ définie plus haut. On veut exécuter

$$\mathbf{P}, \mathbf{D_0} - \ - \vdash \phi.$$

On utilise les correspondances décrites plus haut entre notre syntaxe et les prédicats de requête et de transitions implicitement pour alléger la démonstration car la syntaxe est en elle-même très lisible. Seules les quelques premières étapes de la démonstration sont présentées ici, pour donner un aperçu du fonctionnement de chaque règle du système d'inférence. La démonstration complète est en annexe A.2.

$$
\begin{aligned}
\mathbf{P}, \mathbf{D_0} - \ - \vdash &\texttt{read}(3, \texttt{k\_mem}[stack], 128, ReturnRead) \\
&\otimes \texttt{write}(4, \texttt{k\_mem}[stack], 128, ReturnWrite)
\end{aligned} \tag{6}
$$

On applique la règle 1 *Application de la définition de la transaction.*

$$
\begin{aligned}
\mathbf{P}, \mathbf{D_0} - \ - \vdash &\texttt{GET\_FID\_FOR\_PROC}(\texttt{k\_procs}[procA], 3, Fid) \\
&\otimes max(128, Fid.file.size - Fid.pos, ReqSize) \\
&\otimes \texttt{k\_mem}[stack] \ll Fid.file \\
&\otimes Fid.pos := Fid.pos + ReqSize \\
&\otimes ReturnRead = ReqSize \\
&\otimes \texttt{write}(4, \texttt{k\_mem}[stack], 128, ReturnWrite)
\end{aligned} \tag{7}
$$

On applique la règle 1 *Application de la définition de la transaction.*

$$\begin{aligned}
\mathbf{P}, \mathbf{D_0} - \; - \; \vdash & (\texttt{k_fdesc}[F].procs \ni \texttt{k_procs}[procA] \wedge \texttt{k_fdesc}[F].fd = 3) \\
& \otimes Fid = \texttt{k_fdesc}[F] \\
& \otimes max(128, Fid.file.size - Fid.pos, ReqSize) \\
& \otimes \texttt{k_mem}[stack] \ll Fid.file \\
& \otimes Fid.pos := Fid.pos + ReqSize \\
& \otimes ReturnRead = ReqSize \\
& \otimes \texttt{write}(4, \texttt{k_mem}[stack], 128, ReturnWrite)
\end{aligned} \tag{8}$$

On applique la règle 2 *Requête de la base de données* avec l'unification

$$\begin{aligned}
& [F \mapsto 1] \text{ car} \\
& \mathcal{O}^d(D_0) \models^c \texttt{k_fdesc}[1].procs \ni \texttt{k_procs}[procA] \wedge \texttt{k_fdesc}[1].fd = 3
\end{aligned} \tag{9}$$

$$\begin{aligned}
\mathbf{P}, \mathbf{D_0} - \; - \; \vdash & Fid = \texttt{k_fdesc}[1] \\
& \otimes max(128, Fid.file.size - Fid.pos, ReqSize) \\
& \otimes \texttt{k_mem}[stack] \ll Fid.file \\
& \otimes Fid.pos := Fid.pos + ReqSize \\
& \otimes ReturnRead = ReqSize \\
& \otimes \texttt{write}(4, \texttt{k_mem}[stack], 128, ReturnWrite)
\end{aligned} \tag{10}$$

On applique la règle 2 *Requête de la base de données* avec l'unification

$$\begin{aligned}
& [Fid \mapsto \texttt{k_fdesc}[1]] \text{ car} \\
& \mathcal{O}^d(D_0) \models^c \texttt{k_fdesc}[1] = \texttt{k_fdesc}[1]
\end{aligned} \tag{11}$$

$$\begin{aligned}
\mathbf{P}, \mathbf{D_0} - \; - \; \vdash & max(128, \texttt{k_fdesc}[1].file.size - \texttt{k_fdesc}[1].pos, ReqSize) \\
& \otimes \texttt{k_mem}[stack] \ll \texttt{k_fdesc}[1].file \\
& \otimes \texttt{k_fdesc}[1].pos := \texttt{k_fdesc}[1].pos + ReqSize \\
& \otimes ReturnRead = ReqSize \\
& \otimes \texttt{write}(4, \texttt{k_mem}[stack], 128, ReturnWrite)
\end{aligned} \tag{12}$$

On applique la règle 2 *Requête de la base de données* avec l'unification

$$\begin{aligned}
& [ReqSize \mapsto 128] \text{ car} \\
& \mathcal{O}^d(D_0) \models^c \texttt{k_fdesc}[1].file.size - \texttt{k_fdesc}[1].pos = 128 \\
& \text{car } \mathcal{O}^d(D_0) \models^c p_valueInTable(\texttt{k_fdesc}, 1, pos, 0) \\
& \wedge \mathcal{O}^d(D_0) \models^c p_valueInTable(\texttt{k_fdesc}, 1, file, \texttt{k_fs}[fichier1]) \\
& \wedge \mathcal{O}^d(D_0) \models^c p_valueInTable(\texttt{k_fs}, fichier1, size, 128) \\
& \wedge \mathcal{O}^d(D_0) \models^c max(128, 128, 128)
\end{aligned} \tag{13}$$

$$\begin{aligned}
\mathbf{P}, \mathbf{D_0} - \; - \; \vdash & \texttt{k_mem}[stack] \ll \texttt{k_fdesc}[1].file \\
& \otimes \texttt{k_fdesc}[1].pos := \texttt{k_fdesc}[1].pos + 128 \\
& \otimes ReturnRead = 128 \\
& \otimes \texttt{write}(4, \texttt{k_mem}[stack], 128, ReturnWrite)
\end{aligned} \tag{14}$$

On applique la règle 3 *Mise à jour élémentaire de la base de données*
avec l'unification vide [] car

$\exists \mathbf{D_1}, \mathcal{O}^t(D_0, D_1) \models p\_add\_flow(\texttt{k\_fs}[fichier1], \texttt{k\_mem}[stack], true)$ car

$\mathcal{O}^d(D_0) \models p\_valueInTable(\texttt{k\_fdesc}, 1, file, \texttt{k\_fs}[fichier1])$

d'après la définition 10, $\mathbf{D_1}$ est tel que

$\mathcal{O}^d(D_1) = \mathcal{O}^d(D_0) \cup p\_flow(1, \texttt{k\_fs}[fichier1], \texttt{k\_mem}[stack], true)$

car $\neg \exists N, Source, Dest, Append,$

$\mathcal{O}^d(D_0) \models p\_flow(N, Source, Dest, Append)$ (15)

(16)

...

La transaction poursuit son exécution jusqu'à la fin de l'appel système `write` ; de la même manière que l'état $\mathbf{D_1}$, les états $\mathbf{D_2}$ jusqu'à $\mathbf{D_6}$ sont construits par le système d'inférence à chaque modification de la base de données.

...

Comme toute application du système d'inférence, une fois que toutes les étapes de la transaction sont terminées — le but est entièrement exécuté — la démonstration–exécution termine par l'application de l'unique axiome :

$$\mathbf{P}, \mathbf{D_6} \vdash () \qquad \text{qui est vrai par l'axiome 0.} \tag{17}$$

Conclusion

Démonstration. D'après [4], et plus particulièrement le théorème 4.2 donnant la complétude et la correction du système d'inférence, comme il existe une déduction de ϕ avec la base de transaction $\mathbf{P}$ dont le chemin d'exécution est $\mathbf{D_0}$, $\mathbf{D_1}$, $\mathbf{D_2}$, $\mathbf{D_3}$, $\mathbf{D_4}$, $\mathbf{D_5}$, $\mathbf{D_6}$, on a donc

$$\mathbf{P}, \mathbf{D_0 D_1 D_2 D_3 D_4 D_5 D_6} \models \phi$$

□

3.4.2 Situation finale

L'état final est présenté informellement dans la table 9. L'exécution de la transaction retourne également l'unification pour les variables libres de la transaction : $[ReturnRead \mapsto 128, ReturnWrite \mapsto 128]$.

Deux flux d'information ont été enregistrés dans la table des flux, un premier en lecture du premier fichier vers la zone mémoire utilisée comme pile du processus A, puis un second de cette même zone mémoire vers le second fichier. Un peu plus formellement, voici la liste des prédicats donnée par $\mathcal{O}^d(D_6)$.

— $p\_valueInTable(\texttt{k\_fs}, fichier1, type, f)$;
— $p\_valueInTable(\texttt{k\_fs}, fichier1, uid, \texttt{k\_procs}[procA])$;
— $p\_valueInTable(\texttt{k\_fs}, fichier1, gid, \texttt{k\_procs}[procA])$;

fs	fichier 1	fichier 2
type	f	f
uid	procA	procB
gid	procA	procA
mode	`rw-r---`	`rw-rw--`
size	128	740

(a) Système de fichiers

fdesc	1	2
file	fichier 1	fichier 2
procs	{procA}	{procA}
opts	{READ}	{APPEND}
fd	3	4
pos	128	740

(b) Descripteurs de fichiers

mem	stack	data
size	5000	10000

(c) Mémoire

proc	1
euid	1000
egid	1000
stack_mem	stack

(d) Processus

Ordre	Direction
1	fichier1 $\rightarrow$ stack
2	stack $\rightarrow$ fichier2

(e) Flux

TABLE 9 – Situation finale

— $p\_valueInTable(\texttt{k\_fs}, fichier1, mode, \texttt{rw-r----})$;
— $p\_valueInTable(\texttt{k\_fs}, fichier1, size, 128)$;

— $p\_valueInTable(\texttt{k\_fs}, fichier2, type, f)$;
— $p\_valueInTable(\texttt{k\_fs}, fichier2, uid, \texttt{k\_procs}[procB])$;
— $p\_valueInTable(\texttt{k\_fs}, fichier2, gid, \texttt{k\_procs}[procA])$;
— $p\_valueInTable(\texttt{k\_fs}, fichier2, mode, \texttt{rw-rw----})$;
— $p\_valueInTable(\texttt{k\_fs}, fichier2, size, 740)$;

— $p\_valueInTable(\texttt{k\_fdesc}, 1, file, fichier1)$;
— $p\_valueInTable(\texttt{k\_fdesc}, 1, procs, \{\texttt{k\_procs}[procA]\})$;
— $p\_valueInTable(\texttt{k\_fdesc}, 1, opts, \{READ\})$;
— $p\_valueInTable(\texttt{k\_fdesc}, 1, fd, 3)$;
— $p\_valueInTable(\texttt{k\_fdesc}, 1, pos, 128)$;

— $p\_valueInTable(\texttt{k\_fdesc}, 2, file, fichier2)$;
— $p\_valueInTable(\texttt{k\_fdesc}, 2, procs, \{\texttt{k\_procs}[procA]\})$;
— $p\_valueInTable(\texttt{k\_fdesc}, 2, opts, \{APPEND\})$;
— $p\_valueInTable(\texttt{k\_fdesc}, 2, fd, 4)$;
— $p\_valueInTable(\texttt{k\_fdesc}, 2, pos, 740)$;

— $p\_valueInTable(\texttt{k\_mem}, stack, size, 5000)$;

— $p\_valueInTable(\texttt{k\_mem}, data, size, 10000)$;

— $p\_valueInTable(\texttt{k\_procs}, procA, euid, 1000)$;
— $p\_valueInTable(\texttt{k\_procs}, procA, egid, 1000)$;
— $p\_valueInTable(\texttt{k\_procs}, procA, stack\_mem, \texttt{k\_mem}[stack])$.

— $p\_flow(1, \texttt{k\_fs}[fichier1], \texttt{k\_mem}[stack], true)$;
— $p\_flow(2, \texttt{k\_mem}[stack], \texttt{k\_fs}[fichier2], true)$.

3.5 Conclusion de l'exemple

Cet exemple montre la puissance de la $\mathcal{CTR}$ et de son système d'inférence. À partir de l'état de départ, on peut démontrer *de manière constructive* l'existence d'un chemin menant jusqu'à un état final où la transaction est exécutée entièrement ; et ceci même dans un contexte qui n'est pas celui de la création de la $\mathcal{CTR}$. En particulier, nous arrivons à mettre en évidence l'apparition de flux d'information dans le système et à les caractériser. Mieux encore, ils sont ordonnés et permettent donc d'établir avec certitude entre quels conteneurs l'information a été transférée. Nous constatons que les oracles de données et de transition permettent un tout nouvel usage du formalisme, sans rien modifier de ses propriétés de correction et de complétude [4]. En fait, on peut même grâce au système d'inférence construire tous les chemins permettant d'exécuter la transaction. Il suffit pour cela de changer la règle à exécuter à chaque étape et de tester toutes les unifications possibles. Bien entendu, à part pour démontrer la solidité et la flexibilité, il n'y a aucun intérêt à developper de telles preuves lourdes et fastidieuses manuellement. En revanche, le formalisme est particulièrement approprié pour les outils de raisonnement automatique tels que Prolog. C'est l'idée que nous avons retenu pour notre interpréteur.

4 Implémentation, tests et résultats

4.1 Interpréteur Prolog/C++

Après avoir construit notre modèle théorique en nous appuyant sur la $\mathcal{CTR}$, nous avions besoin d'un interpréteur pour cette logique afin de faire tourner nos tests. Nous avons étudié un interpréteur de la $\mathcal{CTR}$ en Prolog conçu par Amalia Sleghel et Anthony Bonner [19, 20]. Cet interpréteur était développé pour XSB, un interpréteur Prolog avec de très bonnes performances. Nous avons néanmoins préféré travaillé avec SWI-Prolog. En effet, bien que ce dernier présente des performances inférieures, il a d'autres avantages sur XSB, comme le respect plus strict du standard ISO de Prolog et la disponibilité d'une interface plus fournie avec le C++. Nous avons developpé notre interpréteur, baptisé ALFRED, sur la base de l'interpréteur pour la $\mathcal{CTR}$. Nous avons conservé la partie d'analyse syntaxique et d'exécution et nous avons entièrement changé la partie concernant les requêtes et les mises à jour de la base de données. Nous avons également modularisé entièrement l'interpréteur et séparé clairement les responsabilités — compilation du code en syntaxe $\mathcal{CTR}$, moteur d'exécution, interface utilisateur, etc. — en différents fichiers de définitions de prédicats exportant chacun une interface minimale.

La raison de l'utilisation de deux langages est simple : ils ne se plient pas au mêmes usages. Nous avions besoin de la facilité de programmation offerte par Prolog ainsi que ses mécanismes très puissants de raisonnement automatisé, tels que l'unification, pour la construction des résultats. Dans le même temps, Prolog était très mal adapté à la modélisation de la base de données et de ses modifications. En effet, les structures de données offertes par Prolog — listes, séquences de caractères, etc. — sont très primitives et manquent de souplesse dans leur utilisation. Nous avions besoin de la programmation impérative et orientée objet de C++ pour cela. De plus, le C++ nous permet d'instrumenter l'exécution en stockant de nombreuses informations utiles tout au long de

celle-ci afin de compléter les statistiques fournies par Prolog. Nous avons ainsi pu construire des graphes d'exécutions, comme on le verra plus loin.

4.1.1 Organisation du code

Le code mêle C++ et Prolog grâce à l'interface disponible dans SWI-Prolog. Les fichiers constituant l'interpréteur de la $\mathcal{CTR}$ sont présentés ci-dessous. Nous ne détaillons pas ici le code-source en C++ qui comporte de nombreux fichiers. Le lecteur est invité à visiter la documentation attachée au code-source, disponible sur l'Internet [9]. Le code en C++ implémente la base de données et permet des mises à jour « annulables », fonctionnalité au cœur du fonctionnement de la $\mathcal{CTR}$ et nécessaire pour la recherche exhaustive de solutions via le *backtracking*. Lorsque l'interpréteur voit une mise à jour dans la base de données, il retient le moyen de revenir à l'état antérieur. Cette propriété constitue tout l'intérêt de la $\mathcal{CTR}$ et doit être implémentée avec soin. Cependant, grâce au mécanisme des oracles, la base peut être implémentée de manière arbitraire, en-dehors de tout formalisme logique. Les oracles font eux partie du formalisme logique et doivent être capable de manipuler et interroger la base de données. Cela est réalisé dans notre cas grâce à l'interface C++ de SWI-Prolog qui permet de définir des prédicats externes codés en C++, appelables par l'exécuteur Prolog de la même manière que des prédicats normaux, écrits en Prolog.

Architecture d'ALFRED Le répertoire d'installation d'ALFRED comporte les fichiers suivants :

```
interpreter/
├── interpctr_lib/ ..... Code C++ implémentant la base de données
│   └── html/ ..... Documentation du code C++
├── syscalls/ ..... Définitions des appels système
│   └── macros/ ..... Définitions des macros
├── tests/ ..... Définitions des jeux de test
│   ├── tests/*.ctr ..... Jeux de test
│   └── tests/*.db ..... Fichiers d'initialisation de la base de données
├── *.pl ..... Code Prolog de l'interpréteur
└── doc/ ..... Documentation du code Prolog
```

Le code C++ est compilé en une bibliothèque chargeable dans le code Prolog.

Détail du code Prolog

interpctr.pl module prolog constituant l'interface utilisateur, il permet de compiler des bases de transactions, de charger des états de la base de données et de lancer l'exécution de transactions ;

ctr.pl module prolog constituant le cœur de l'interpréteur, exécute les transactions ;

compiler.pl module prolog servant à transformer les bases de transactions écrites en utilisant le formalisme de la CTR en syntaxe Prolog ;

parser.pl sous-module Prolog du compilateur servant à interpréter les opérateurs de la CTR et à reconstituer les termes Prolog correspondants,

queries.pl module Prolog implémentant l'oracle de données ;

updates.pl module Prolog implémentant l'oracle de transitions.

4.1.2 Mécanisme de compilation

On distingue trois types de fichiers écrits en syntaxe $\mathcal{CTR}$ pour ALFRED :

— Les fichiers `.ctr` dans `syscalls/` sont les appels système. Il y a un fichier par appel système et il porte son nom. Il contient les différentes clauses définissant cet appel système et éventuellement des sous-prédicats.

— Les fichiers `.ctr` dans `syscalls/macros/` sont les macros, des bouts de code $\mathcal{CTR}$ utilisés par plusieurs appels systèmes et factorisés en une clause.

— Les fichiers `.ctr` dans `tests/` sont des scénarios de tests. Ils contiennent quelques prédicats, dont un principal, constituant le point d'entrée du scénario de test.

Le processus de compilation inclut une phase de *preprocessing*. Les fichiers de test peuvent inclure des définitions d'appels système, qui eux-mêmes peuvent inclure des définitions de macros. Lorsque la compilation d'un fichier de test est demandée, toutes les inclusions sont résolues et le résultat est un seul fichier avec toutes les définitions de macros et d'appels système requises suivies des tests. C'est sur ce fichier que l'on effectue la compilation proprement dite. Cette dernière consiste à réécrire le fichier en syntaxe Prolog. Les termes et constructions Prolog utilisés sont propres à ALFRED — et non décrits dans sa spécification. La compilation ne comprend pas d'optimisation dans l'état actuel du prototype. Le processus complet s'achève par le chargement dans Prolog de tous les prédicats générés.

4.2 Exécution des tests

4.2.1 Interface utilisateur

Dans l'état actuel de notre prototype, l'interface est minimaliste et en console uniquement. Nous utilisons l'interpréteur SWI-Prolog. Nous chargeons les fichiers de l'interpréteur de la logique transactionnelle concurrentielle avec la première ligne. Ensuite, le prédicat `ctr_comp/1` prend en paramètre un fichier programme (sans l'extension) et lance le processus de compilation décrit plus haut. Il en résulte un fichier de même nom avec l'extension `.o` dont les prédicats sont chargés et peuvent être exécutés. On peut initialiser la base de données à l'aide du prédicat `load_db_state/2`. Elle prend en paramètre le fichier contenant les prédicats d'initialisation et une variable qui sera initialisée avec le processus effectuant les appels système. Pour le moment, seule cette version existe mais si plusieurs processus étaient nécessaires au démarrage du scénario de test, le prédicat prendrait en second paramètre une liste de processus. Ensuite, on peut lancer l'exécution du prédicat principal de test avec le prédicat `execc/1`, qui est l'interpréteur proprement dit.

Grâce à Prolog, il est possible d'interagir avec l'interpréteur. Par exemple, le prédicat `display/0` affiche le contenu des tables. Les prédicats `graph/1` et `nograph/0` servent respectivement à activer la sortie sous forme de graphique — `graph/1` prend en paramètre un nom de fichier de sortie — et à la désactiver. L'annexe A.3 présente une session $\mathcal{CTR}$ complète.

4.2.2 Principe d'utilisation des tests

Notre objectif à terme est de pouvoir générer, de manière la plus automatisée possible, des tests pour `kblare`. Concrètement, nous souhaitons pouvoir mettre le système surveillé par `kblare` dans un certain état de départ, répliquer cet état dans notre modèle et faire tourner le scénario à la fois dans le système réel et dans le modèle pour en comparer les sorties.

Un scénario de test est constitué de quelques actions effectuées par des processus, correspondant

à un cas de propagation de tags spécifique. À la fin du test, dans le système réel surveillé par `kblare`, nous obtenons de nouveaux tags pour les objets du système (fichiers ou processus). Du côté de notre modèle, nous obtenons un ensemble de séquences de flux possibles. Pour chaque séquence, on peut calculer les tags correspondants. On conclut que la sortie de `kblare` est correcte s'il produit des tags valides, c'est-à-dire faisant partie de la sortie de notre modèle. Dans le cas contraire, les tags produits ne font pas partie de l'ensemble des ensembles de tags théoriquement possible. On peut donc conclure que `kblare` présente une erreur (à moins que ce ne soit la modélisation qui soit incomplète).

4.3 Exemple client–serveur

L'exemple client–serveur (programme 1 page 31) présente la situation suivante. Un processus d'un serveur web reçoit une requête d'un usager pour téléverser un fichier. Il ouvre donc deux fichiers : un fichier de données source en lecture seule (par exemple une base de données d'usagers) et un deuxième fichier en écriture seule (le tampon qui va recevoir le fichier que l'utilisateur connecté au serveur cherche à téléverser sur ce dernier). Il lit la base de données pour la charger en mémoire. Avant de commencer à répondre à la requête, il *forke* pour pouvoir écouter sur le port web et répondre à une éventuelle requête qui arriverait avant que la première ne soit entièrement terminée [3].

Les deux processus s'exécutent donc en parallèle. Le deuxième processus reçoit justement une requête. C'est un autre usager qui souhaite téléverser un fichier sur le serveur. Comme, pour justifier l'intérêt de l'exemple, notre serveur est très mal codé, les deux processus utilisent malheureusement le même tampon pour écrire leur fichier, de sorte que l'un d'entre eux va être écrasé. Les deux processus terminent leur exécution en fermant leurs descripteurs de fichier. Selon l'entrelacement de ces différentes opérations, l'état du système peut être quelconque, et `kblare` devrait calculer des tags différents pour chaque situation résultante.

4.3.1 Explication du code de l'exemple

Le code débute par la déclaration des appels système utilisés. Dans ce cas, cinq définitions, accompagnées de leurs macros, sont incluses. Ensuite se trouvent trois clauses. La clause principale est `run`. Elle prend en paramètre un processus (qui est un index valide dans la table `k_procs`), et trois variables de résultat, qui seront unifiées aux retours des trois appels système de l'exemple, à savoir `read` par le premier processus puis `write` par les deux processus. Nous avons fait cette distinction entre les clauses pour séparer la première partie, qui est entièrement séquentielle — un seul processus est à l'œuvre de la seconde qui est une exécution concurentielle de deux processus. La clause `run` consiste en l'exécution de la clause `prepare` suivie de celle de la clause `runboth`.

4.3.2 Clause `prepare`

La clause `prepare` crée les descripteurs de fichiers (les entrées de la table `k_fdesc`). Ensuite, le processus effectue l'appel système `read`. Enfin, le deuxième processus est créé par *fork* du premier. Dans cette clause, il s'agit d'une simple séquence, comme on le voit à travers l'usage des opérateurs $\otimes$; la sémantique est donc très simple. Le premier paramètre de cette clause est l'identifiant du

3. Ce comportement est très courant dans les serveurs web. Un processus écoute en boucle sur un port pour recevoir les requêtes et pour chacune, lance un nouveau processus — ou dans d'autres implémentations, juste un nouveau fil d'exécution (un *thread*) — pour y répondre.

syscall(`read`)
syscall(`write`)
syscall(`open`)
syscall(`fork`)
syscall(`close`)
run(ProcA, ReadA, WriteA, WriteB) $\leftarrow$
    prepare($ProcA, ProcB, Fd1, Fd2, ReadA$)
    $\otimes$ runboth($ProcA, ProcB, Fd1, Fd2, WriteA, WriteB$).
prepare(ProcA, ProcB, Fd1, Fd2, ReadA) $\leftarrow$
    $StackA := k\_procs[ProcA].stack\_mem$
    $\otimes \odot$[`open`($ProcA,' source', [o\_rdonly], \_, Fd1$)]
    $\otimes \odot$[`open`($ProcA,' tampon', [o\_wronly], \_, Fd2$)]
    $\otimes \odot$[`read`($ProcA, Fd1, StackA, 128, ReadA$)]
    $\otimes \odot$[`fork`($ProcA, fork(0, ProcB)$)].
runboth(ProcA, ProcB, Fd1, Fd2, WriteA, WriteB) $\leftarrow$
    (
        $StackA := k\_procs[ProcA].stack\_mem$
        $\otimes \odot$[`write`($ProcA, Fd2, StackA, 128, WriteA$)]
        $\otimes \odot$[`close`($ProcA, Fd1, 0$)]
        $\otimes \odot$[`close`($ProcA, Fd2, 0$)]
) |
    (
        $StackB := k\_procs[ProcB].stack\_mem$
        $\otimes \odot$[`write`($ProcB, Fd2, StackB, 56, WriteB$)]
        $\otimes \odot$[`close`($ProcB, Fd1, 0$)]
        $\otimes \odot$[`close`($ProcB, Fd2, 0$)]
).

Programme $\mathcal{CTR}$ 1 – Programme d'exemple client–serveur

premier processus. Les quatre autres sont des variables de résultat. La première, $ProcB$, contient l'index de la table `k_procs` correspondant au second processus, issue du *fork*. $Fd1$ et $Fd2$ contiennent les index de la table `k_fdesc`, ce sont les descripteurs de fichiers attachés aux fichiers 1 et 2. Enfin, la dernière variable est le résultat de l'appel système `read`.

La première action de la clause est $StackA := k\_procs[ProcA].stack\_mem$. Ceci est juste une syntaxe un peu particulière pour exprimer l'unification de la variable libre $StackA$ avec le champ $stack\_mem$ de l'entrée $ProcA$ de la table `k_procs`. Ensuite, quatre appels système suivent. On remarque qu'ils sont placés en isolation avec la modalité $\odot$. Ceci n'est pas strictement nécessaire étant donné que l'exécution est de toute façon séquentielle. L'utilité réside uniquement dans l'uniformisation de la syntaxe avec le reste du programme.

4.3.3 Clause `runboth`

La clause `runboth` correspond aux écritures en parallèle des deux processus dans le même fichier, le fichier 2. La clause reçoit en paramètre quatre variables unifiées et deux variables de résultat, non unifiées. Les premières correspondent aux deux processus et aux deux descripteurs de fichiers — des entrées dans les tables `k_procs` et `k_fdesc` respectivement ; les deux dernières aux résultats des appels système `write`. La clause est divisée deux blocs correspondant aux deux processus s'exécutant en parallèle. Ces deux blocs sont entre parenthèses et séparés par l'opérateur de conjonction concurrentielle |. Dans chacun, l'exécution est entièrement séquentielle et les actions sont séparées par l'opérateur de conjonction séquentielle $\otimes$. L'exécution consiste en un appel système `write` et deux appels `close` pour chacun des deux descripteurs de fichiers. Comme dans la clause précédente, les appels système sont exécutés en isolation. Ainsi, lorsqu'un processus entame un appel système comme `write`, il le mène jusqu'à son terme sans que l'autre processus ne puisse entamer d'exécution. L'intérêt de la $\mathcal{CTR}$ est de pouvoir nous permettre de gérer assez finement le degré d'atomicité que l'on choisit. En effet, nous pourrions vouloir voir s'intercaler les instructions des appels système plutôt que les appels système eux-mêmes. Nous avons fait le choix de considérer que les appels étaient atomiques. C'est contestable car dans la réalité, il arrive que des appels système soient interrompus (parce qu'ils sont en attente d'une ressource, typiquement). Cependant, bien que notre formalisme soit prévu pour cela, c'est notre modélisation du code des appels système qui ne nous permet pas de gérer ce niveau de détail. Une instruction unitaire d'un appel système ne correspond pas toujours exactement à une action unitaire dans notre modèle. Notre approche reste réaliste car dans un véritable système d'exploitation, les appels système s'assurent de verrouiller toutes les structures de données qu'ils utilisent, empêchant donc les autres appels système d'interférer avec leur exécution.

4.4 Temps d'exécution

Pour mesurer l'efficacité de l'interpréteur et quantifier le problème de passage à l'échelle, nous avons mesuré le temps d'exécution d'un même scénario avec plus ou moins de processus en parallèle. Les statistiques sont récoltées par Prolog à l'aide du prédicat `time/0`. Un exemple d'utilisation est le but Prolog suivant :

```
?- time(aggregate_all(count,execc(run(0)),C)).
% 4,477 inferences, 0,004 CPU in 0,004 seconds (99% CPU, 1149874 Lips)
C = 1.
```

Le tableau 10 donne les résultats pour l'exécution de l'exemple client–serveur avec un nombre de processus variables. Les tests ont été réalisés sur une installation Gentoo (GNU/Linux), avec GCC

Situation	Nombre de solutions	Inférences	Temps total (s)	Lips
1 processus	1	2 510	0,002	1 033 578
2 processus	70	69 645	0,037	1 887 043
3 processus	34650	28 988 799	15,401	1 886 135
4 processus	< non terminé >	8 459 640 928	6076,753	1 392 135

TABLE 10 – Statistiques sur l'exécution de l'exemple client–serveur dans différentes configurations. L'exécution avec quatre processus a été interrompue par manque de mémoire.

4.7.3 et SWI-Prolog 6.4.1 (patché par nos soins), sur un Intel Core i3 cadencé à 2,13 GHz, disposant de 4 Go de mémoire vive. Notre code n'est pas exécuté de manière parallèle et utilise donc un seul cœur. Dans l'exemple client–serveur, un processus ouvre deux fichiers, fait une lecture du premier, puis forke. Dans chaque processus-fils ainsi que dans le processus-père, une opération d'écriture a lieu puis les deux descripteurs de fichiers sont fermés. Pour n processus, il y a donc un total de 3 (appels système effectués par le premier processus) $+\,n-1$ (fork) $+\,3n$ (appels système effectués par les processus en parallèle) appels système, dont $3n$ peuvent s'entrelacer. On prend comme paramètre la quantité de processus au total, le processus-père étant compris. Les statistiques données sont les suivantes pour chaque configuration :

Nombre de solutions Nombre d'entrelacements d'appels système calculé par Prolog ;

Inférences Nombre de buts que Prolog a résolus ;

Temps total Le temps d'exécution en secondes pour l'exécution ;

Lips Nombre d'inférences logiques par secondes.

Pour un nombre limité de processus, les temps d'exécutions sont très raisonnables. En revanche, ils présentent une croissance difficilement contrôlables dès que de nombreux appels système peuvent être intercalés. L'explosion combinatoire est donc notre principal problème. Cependant, notre utilisation première de notre modèle est le test unitaire, ce qui implique de très petits scénarios de tests ne mettant en jeu qu'un nombre minimal de processus. Dans le cadre de cet usage, les performances sont acceptables. En ce qui concerne l'utilisation de la mémoire, elle suit à peu près la même croissance que le temps de calcul car Prolog doit maintenir beacoup d'état pour pouvoir explorer exhaustivement toutes les solutions. De plus, nous suspectons l'interface C++ de SWI-Prolog — ou notre code...— de présenter des fuites mémoire. En effet, nous avons noté lors de nos tests que de la mémoire restait allouée dans le tas après la fin de l'exécution de scénarios de tests jusqu'à la terminaison de l'exécuteur Prolog alors même que le ramasse-miettes de Prolog est censée la libérer au fur et à mesure.

4.5 Multigraphe d'exécution

Notre majeur problème est la performance des outils que nous utilisons. Même si nous n'utilisons que de petits scénarios de test — car notre objectif est le test unitaire — le nombre d'entrelacements de processus croit exponentiellement avec le nombre de processus en jeu dans chaque test et le nombre d'appels système qu'ils réalisent. Pour pouvoir interpréter les résultats, nous nous appuyons sur les résultats des travaux de recherches relatifs à l'algèbre des processus pour en donner une représentation claire.

Étant donné que nous étions principalement intéressés par les flux causés par les appels système pour évaluer le bon fonctionnement de `kblare`, nous avons choisi une représentation sous forme

de graphe qui fait apparaître clairement les flux tout en faisant abstraction du reste. Pour chaque scénario de tests, nous obtenons un multigraphe. Chaque nœud représente soit un flux, soit une séquence d'actions ne comportant pas de flux. Les arcs comportent une valuation, représentée par une couleur. Le multigraphe est une représentation des multichemins d'exécution de la $\mathcal{CTR}$: chaque chemin ne comportant que des arcs valués identiquement (c'est-à-dire de la même couleur) correspond à une séquence de flux possible. Pour rappel, à partir de cette séquence de flux, il est possible de calculer un ensemble de tags de `kblare`, et de comparer cet ensemble à celui donné par `kblare`.

Définition 11 (Multigraphe d'exécution)**.** Un multigraphe d'exécution $\mathfrak{G}$ est un quintuplet $\langle V, E, X, \gamma, \text{ord}\rangle$.

- V est l'ensemble des sommets de $\mathfrak{G}$
- E est l'ensemble des arcs de $\mathfrak{G}$
- X est l'ensemble des exécutions de $\mathfrak{G}$
- γ est la fonction d'incidence de $\mathfrak{G}$
  $\gamma : E \to V \times V \times X$
- ord est la fonction d'ordre des sommets de $\mathfrak{V}$
  $\text{ord} : V \to \mathbb{N}$

$\gamma(g1, g2, x) \in E$, avec $(g1, g2) \in V \times V$ et $x \in X$ tels que $\text{ord}(g1) = n$ signifie que dans l'exécution x, il existe une transition de $g1$ vers $g2$ à la n-ième étape. On a $\text{ord}(g2) = n + 1$.

Un tel multigraphe est présenté dans la figure 2, page 35. L'intérêt de la représentation sous la forme de graphe est qu'elle permet de ne représenter qu'une seule fois toutes les exécutions qui donnent le même ensemble de flux. Nous nous servons de la séquence de flux comme d'une relation d'équivalence entre exécutions pour minimiser l'espace de calcul des comparaisons avec les résultats de `kblare`.

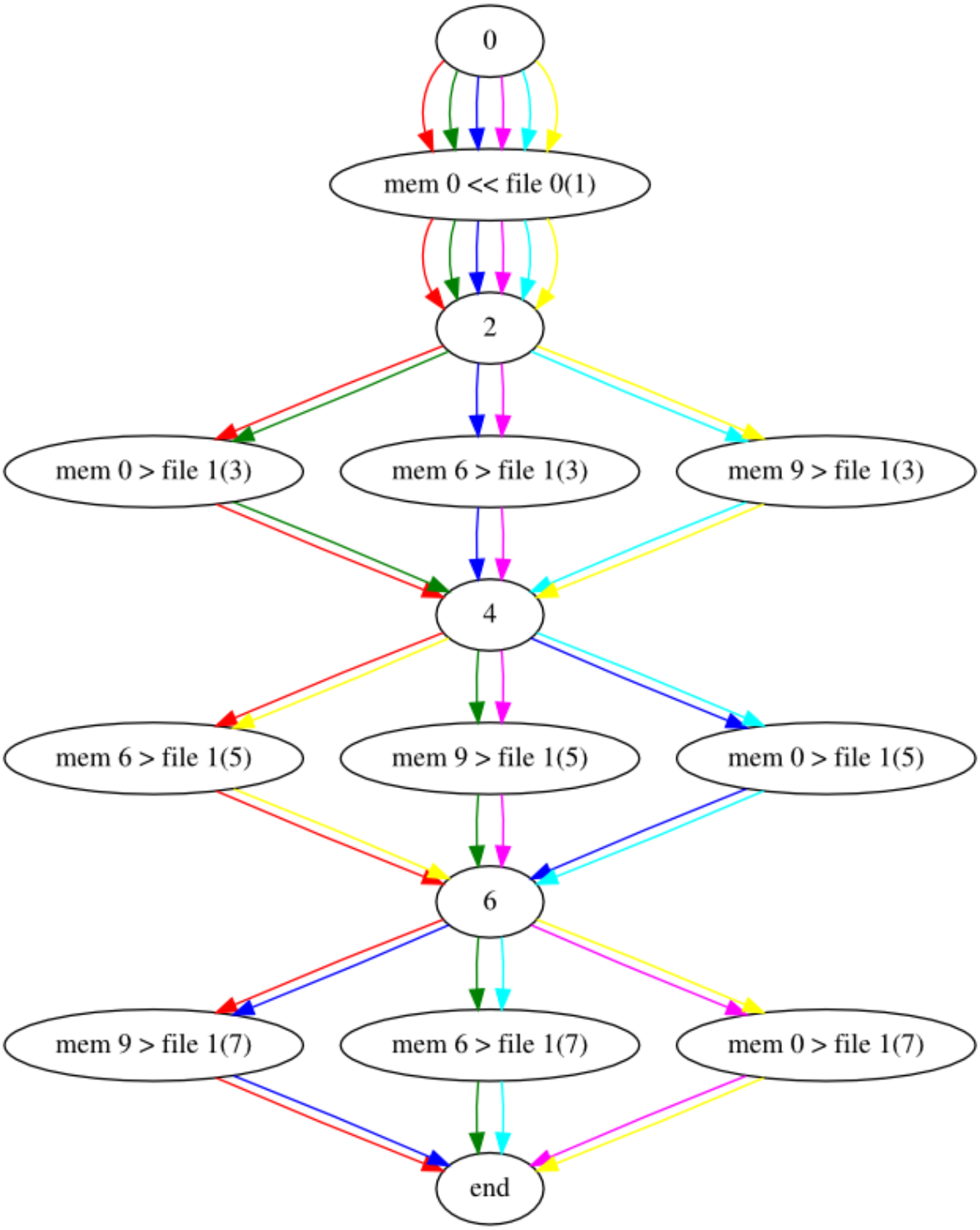

FIGURE 2 – Multigraphe d'exécution pour le programme d'exemple client–serveur avec trois processus

5 Conclusion

5.1 Apport de nos travaux

Nous avons construit une approche complète pour exprimer une sémantique exécutable des appels système dans un système d'exploitation de type UNIX. L'utilisation d'un formalisme logique — la logique transactionnelle concurrentielle — pour représenter des mises à jour n'est pas une découverte nouvelle mais à notre connaissance, c'est la première fois qu'une approche de ce genre est proposée pour décrire un noyau de système d'exploitation en faisant abstraction de l'exécution des processus du système. Bien que les performances soient critiquables, la programmation logique montre une grande flexibilité dans son usage et dans les sorties qu'elle peut produire. Les flux d'information sont exprimés entre la mémoire des processus ou les fichiers et les différentes exécutions possibles sont présentées et classées en fonction des flux engendrés. Nous pouvons grâce à cette sémantique construire des jeux de tests unitaires pour tester kblare, ce qui était l'objectif final de notre approche. Nous avons donc prouvé la faisabilité de cette dernière et établi un prototype fonctionnel, base de futurs travaux sur le sujet. De nombreuses améliorations sont encore susceptibles de pouvoir être apportées, en particulier pour maîtriser l'explosion combinatoire inévitable dans nos jeux de tests faisant intervenir plus de deux ou trois processus. Une future direction de travail, enfin, est la construction d'une interface plus conviviale pour la construction des tests. Le but final serait d'aboutir à une production au moins semi-automatisée des jeux de tests.

5.2 Travaux futurs

5.2.1 Bisimulation

Pour lutter contre l'explosion combinatoire du nombre d'états atteignables lors de l'exécution concurrente de nombreux processus, les approches explorées par les travaux portant sur la vérification de modèles nous semblent particulièrement opportunes. En effet, de nombreux travaux ont été entrepris dans le domaine des systèmes concurrents pour proposer des méthodes de vérification de leur bon fonctionnement. Comme dans notre cas, cela nécessite de pouvoir gérer un nombre très important de traces possibles. Une technique développée et mise au point est la minimisation par bisimulation [13].

Informellement, on dit de deux graphes qu'ils sont fortement bi-similaires si dans chaque état de chacun d'eux, on peut effectuer les mêmes transactions et aboutir dans des états où là encore, les mêmes transitions sont disponibles. Cela permet de réduire considérablement le nombre de nœuds dans le graphe en établissant une relation d'équivalence entre les deux graphes. L'intérêt pour nous d'une telle minimisation serait évident : éviter de parcourir de nombreuses possibilités d'exécution de scénarios qui génèrent des séquences de flux identiques. En effet, bien souvent l'ordre dans lequel les actions de deux processus s'exécutant en parallèle n'est pas important, en particulier si les actions n'ont pas d'effet l'une sur l'autre.

Dans l'état actuel de développement de notre prototype, nous exploitons partiellement cette idée. Nous explorons toutes les exécutions possibles et nous les classons ensuite en fonction de l'ordonnancement des flux qu'elles causent, sous la forme d'un graphe. La figure 3 montre sur un exemple similaire au programme client–serveur trois exécutions possibles d'un même programme. Ce dernier comporte trois flux, le premier est identique dans les trois exécutions tandis que l'ordre des deux derniers est inversé entre la première exécution et les deux autres. Les étapes 2 et 4 sur la figure montrent une séquence d'actions ne comportant pas de flux. Ces séquences sont peut-être

différentes (elles le sont probablement dans le cas des deuxième et troisième exécution, puisqu'elles constituent deux exécutions différentes) mais la caractérisation sémantique importante ici est qu'elles sont indifférentiables du point de vue des flux produits, ce qui est important dans notre cas. Le multigraphe de la figure 4 est donc bisimilaire à celui de la figure 3 : il présente les mêmes transitions pour les mêmes états.

Le processus de constructions du graphe est pour le moment inclus dans l'interpréteur. Cette méthode montre un problème évident pour passer à l'échelle, lorsque le nombre de processus concurrents dans nos jeux de tests augmente. Un axe de recherche pour nous permettre cette montée en charge serait d'adapter les outils développés pour la vérification de modèle. En effet, de nombreux travaux ont montré qu'il était tout à fait faisable de manipuler des automates de milliers d'états modélisant des processus concurrents avec des outils tels que CADP *(Construction and Analysis of Distributed Processes)* [8], conçu pour la représentation et le traitement de systèmes distribués très vastes. Nous n'avons pas, dans le cadre de ces travaux, poussé plus avant notre approche dans cette direction mais nous avons réalisé une revue des fonctionnalités et performances des outils disponibles.

5.2.2 Heuristiques pour optimiser le travail de l'interpréteur

Dans l'état actuel de notre prototype, étant donné une base de prédicats — l'ensemble des appels système et macros inclus dans le fichier de test — et un prédicat de test, l'interpréteur doit explorer toutes les clauses dans tous les ordres possibles afin de découvrir un chemin d'exécution menant à un état final et à la résolution du test. Cependant, notre objectif premier est de tester que la sortie de `kblare` est correcte, donc nous pouvons collecter des informations de l'exécution de `kblare` pour « guider » notre interpréteur.

Notre outil principal serait la sortie d'un intercepteur d'appel système tel que `strace`. Cet outil peut suivre l'exécution d'un processus et affiche chaque appel système effectué par celui-ci, avec ses paramètres et sa valeur de retour. Cette valeur de retour est très utile car elle permet de savoir si l'appel a réussi d'une part et d'autre part, s'il a échoué, de savoir pour quelle cause. Comme en général, dans notre modèle, nous définissons pour chaque appel système une clause différente par résultat de l'appel système, cette méthode nous permettrait d'éviter l'évaluation inutile de nombreuses clauses. Ensuite, dans le cas où plusieurs processus doivent s'exécuter en parallèle, nous pouvons lancer un traçage `strace` sur tous à la fois. En plus des informations précédentes, nous pourrions fusionner les sorties en fonction de l'horodatage et rétablir un ordre temporel presque total sur les appels système effectués par les différents processus en exécution. De la sorte, nous pourrions réduire drastiquement l'espace de recherche des différentes exécutions possibles. Cette approche semble prometteuse mais elle ne permettra toutefois pas de linéariser entièrement l'exécution car les appels système ne s'exécutent pas de manière tout à fait atomique, et encore moins instantanément, donc l'horodatage est une heuristique délicate à manipuler.

5.3 Perspectives

L'idée de formaliser le comportement des systèmes d'exploitation n'est pas nouvelle mais notre approche l'est en ceci que nous nous intéressons non pas à un système d'exploitation construit *dans le but* de pouvoir en démontrer des propriétés mais à un système d'exploitation pré-existant et largement distribué. Nous pensons donc que nos travaux pourront avoir des applications variés, y compris hors du thème de la sécurité informatique. De plus, s'intéresser à une sémantique des appels

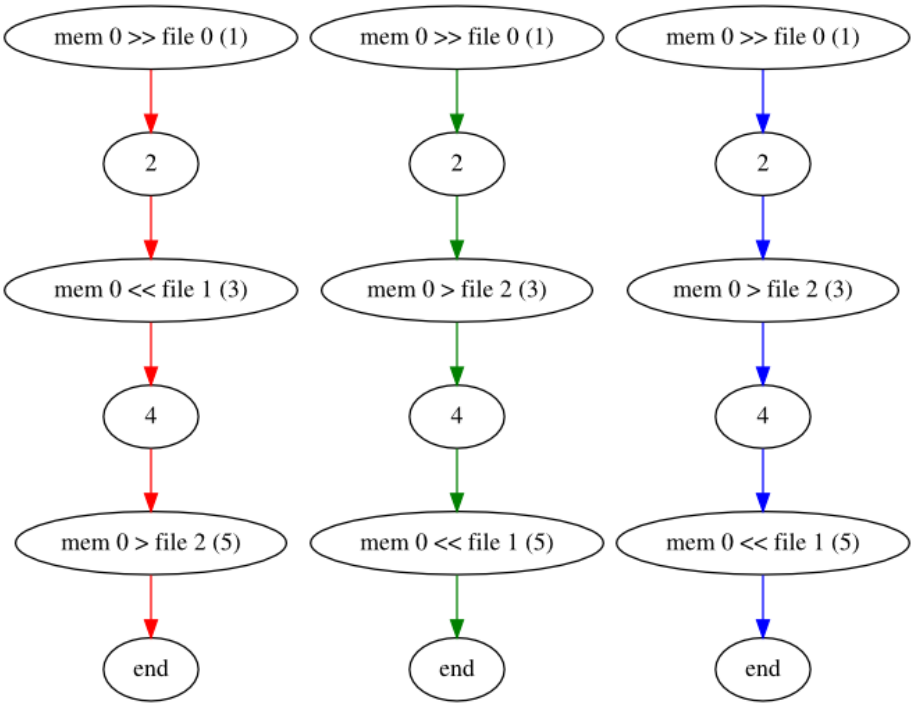

FIGURE 3 – Trois exécutions possibles

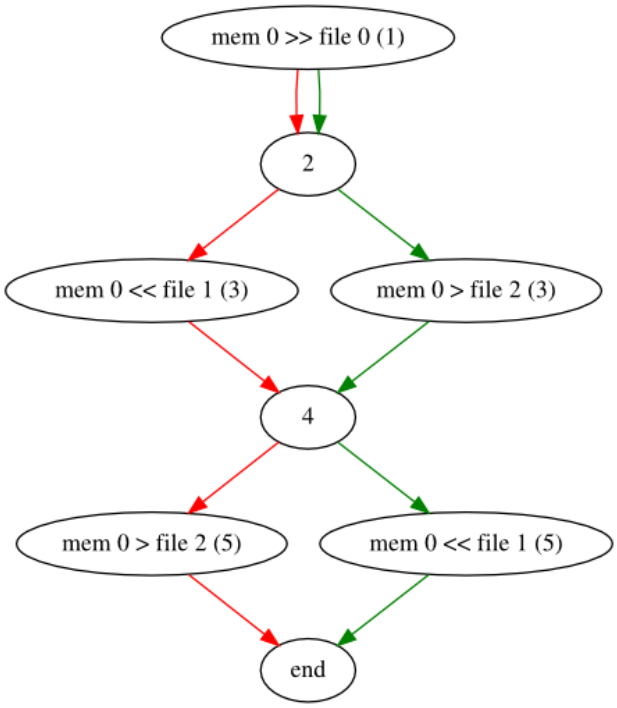

FIGURE 4 – Le multigraphe d'exécution, bisimilaire au graphe précédent

système, qui constitue l'unique interface entre l'espace utilisateur et l'espace noyau sur l'ensemble des systèmes inspirés d'UNIX pourrait fournir des pistes de recherche intéressantes pour dégager des motifs récurrents dans leur exécutions, qui ne seraient pas immédiatement visibles dans leur code, ou encore pour produire de meilleurs tests de non-régression. Nous avons également construit pour du code d'un noyau de système d'exploitation une sémantique à un niveau d'abstraction plus élevé que le langage d'implémentation lui-même. Enfin, nous nous sommes intéressés à la représentation du *noyau* du système d'exploitation tandis que de nombreuses autres approches en font abstraction et s'attachent à décrire les comportements des processus. Là encore, de nouveaux travaux pourraient s'appuyer sur cette idée.

Références

[1] Sandrine Blazy et Xavier Leroy : Mechanized Semantics for the Clight Subset of the C Language. *Journal of Automated Reasoning*, 43(3):263–288, 2009.

[2] Anthony J Bonner et Michael Kifer : Transaction Logic Programming. *In Proceedings of the International Conference on Logic Programming*, pages 257–279, 1993.

[3] Anthony J. Bonner et Michael Kifer : Transaction Logic Programming (or, A Logic of Procedural and Declarative Knowledge). Rapport technique CSRI-323, University of Toronto, novembre 1995.

[4] Anthony J. Bonner et Michael Kifer : Concurrency and Communication in Transaction Logic. *In Joint Intl. Conference and Symposium on Logic Programming*, pages 142–156. MIT Press, 1996.

[5] Guillaume Brogi : Blare. `https://www.blare-ids.org`, 2014. [en ligne, consulté le 3/06/2014].

[6] Alain Colmerauer et Philippe Roussel : History of Programming languages — II. *In* Thomas J. Bergin, Jr. et Richard G. Gibson, Jr., éditeurs : *History of Programming languages*, chapitre The Birth of Prolog, pages 331–367. ACM, New York, NY, USA, 1996.

[7] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel et Anmol N. Sheth : TaintDroid : An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. *In Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[8] Hubert Garavel, Frédéric Lang, Radu Mateescu et Wendelin Serwe : CADP 2010 : A Toolbox for the Construction and Analysis of Distributed Processes. *In* Parosh Aziz Abdulla et K. Rustan M. Leino, éditeurs : *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6605 de *Lecture Notes in Computer Science*, pages 372–387. Springer Berlin Heidelberg, 2011.

[9] Laurent Georget : ALFRED, interpréteur de la logique transactionnelle concurrentielle pour la sémantique des appels système. `http://www.lgeorget.eu/main/alfred/`, 2014. [en ligne, consulté le 2/06/2014].

[10] Christophe Hauser : *Détection d'intrusion dans les systèmes distribués par propagation de teinte au niveau noyau.* Thèse de doctorat, Matisse, Rennes, France, Juin 2013.

[11] Maxwell Krohn et Eran Tromer : Noninterference for a Practical DIFC-Based Operating System. *In Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, SP '09, pages 61–76, Washington, DC, USA, 2009. IEEE Computer Society.

[12] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler et Robert Morris : Information Flow Control for Standard OS Abstractions. *SIGOPS Oper. Syst. Rev.*, 41(6):321–334, oct 2007.

[13] Robin Milner : *Communication and Concurrency.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[14] Robin Milner, Joachim Parrow et David Walker : A Calculus of Mobile Processes, I & II. *Information and Computation*, 100(1):1–77, septembre 1992.

[15] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao et Gerwin Klein : seL4 : From General Purpose to a Proof of Information Flow Enforcement. *In Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 415–429, Washington, DC, USA, 2013. IEEE Computer Society.

[16] V. A. Nepomniaschy, I. S. Anureev, I. N. Mikhailov et A. V. Promskii : Towards Verification of C Programs. C-Light Language and Its Formal Semantics. *Program. Comput. Softw.*, 28(6):314–323, nov 2002.

[17] Nikolaos S. Papaspyrou : A formal semantics for the C programming language. *Doctoral Disseration. National Technical University of Athens. Athens (Greece)*, 15, 1998.

[18] Davide Sangiorgi et David Walker : *PI-Calculus : A Theory of Mobile Processes*. Cambridge University Press, New York, NY, USA, 2001.

[19] Amalia F Sleghel : *An Optimizing Interpreter for Concurrent Transaction Logic*. Thèse de doctorat, University of Toronto, 2000.

[20] Amalia F Sleghel : An Optimizing Interpreter for Concurrent Transaction Logic. `http://www.cs.toronto.edu/~bonner/ctr/`, 2000. [en ligne, consulté le 24/04/2014].

[21] Mingyi Zhao et Peng Liu : Modeling and checking the security of difc system configurations. *In* Ehab Al-Shaer, Xinming Ou et Geoffrey Xie, éditeurs : *Automated Security Management*, pages 21–38. Springer International Publishing, 2013.

A Annexes

A.1 Liste des appels système écrits dans la syntaxe $\mathcal{CTR}$

Les appels système suivants ont été étudiés et écrits avec leurs macros dans la syntaxe de la $\mathcal{CTR}$, et peuvent être utilisés par l'interpréteur.

- `close` ;
- `exec` ;
- `fork` ;
- `open` ;
- `read` ;
- `write`.

Les appels système suivants ont été étudiés et écrits dans la syntaxe de la $\mathcal{CTR}$, mais ne sont pas écrits pour l'interpréteur dans l'état actuel du prototype car ils ne sont utilisés pour aucun exemple jusqu'alors.

- `access` ;
- `lseek` ;
- `getpid` ;
- `getppid` ;
- `chmod` ;
- `chown` ;
- `getuid` ;
- `getgid` ;
- `geteuid` ;

— `getegid`;
— `setuid`;
— `setgid`;
— `seteuid`;
— `setegid`;
— `umask`.

A.2 Démonstration complète utilisant le système d'inférence

$$\begin{aligned}\mathbf{P}, \mathbf{D_0} - \ - \vdash & \texttt{read}(3, \texttt{k\_mem}[stack], 128, ReturnRead) \\ & \otimes \texttt{write}(4, \texttt{k\_mem}[stack], 128, ReturnWrite)\end{aligned} \tag{18}$$

On applique la règle 1 *Application de la définition de la transaction.*

$$\begin{aligned}\mathbf{P}, \mathbf{D_0} - \ - \vdash & \texttt{GET\_FID\_FOR\_PROC}(\texttt{k\_procs}[procA], 3, Fid) \\ & \otimes max(128, Fid.file.size - Fid.pos, ReqSize) \\ & \otimes \texttt{k\_mem}[stack] \ll Fid.file \\ & \otimes Fid.pos := Fid.pos + ReqSize \\ & \otimes ReturnRead = ReqSize \\ & \otimes \texttt{write}(4, \texttt{k\_mem}[stack], 128, ReturnWrite)\end{aligned} \tag{19}$$

On applique la règle 1 *Application de la définition de la transaction.*

$$\begin{aligned}\mathbf{P}, \mathbf{D_0} - \ - \vdash & (\texttt{k\_fdesc}[F].procs \ni \texttt{k\_procs}[procA] \wedge \texttt{k\_fdesc}[F].fd = 3) \\ & \otimes Fid = \texttt{k\_fdesc}[F] \\ & \otimes max(128, Fid.file.size - Fid.pos, ReqSize) \\ & \otimes \texttt{k\_mem}[stack] \ll Fid.file \\ & \otimes Fid.pos := Fid.pos + ReqSize \\ & \otimes ReturnRead = ReqSize \\ & \otimes \texttt{write}(4, \texttt{k\_mem}[stack], 128, ReturnWrite)\end{aligned} \tag{20}$$

On applique la règle 2 *Requête de la base de données* avec l'unification

$$[F \mapsto 1] \text{ car} \tag{21}$$

$$\mathcal{O}^d(D_0) \models^c \texttt{k\_fdesc}[1].procs \ni \texttt{k\_procs}[procA] \wedge \texttt{k\_fdesc}[1].fd = 3$$

$$\begin{aligned}\mathbf{P}, \mathbf{D_0} - \ - \vdash & Fid = \texttt{k\_fdesc}[1] \\ & \otimes max(128, Fid.file.size - Fid.pos, ReqSize) \\ & \otimes \texttt{k\_mem}[stack] \ll Fid.file \\ & \otimes Fid.pos := Fid.pos + ReqSize \\ & \otimes ReturnRead = ReqSize \\ & \otimes \texttt{write}(4, \texttt{k\_mem}[stack], 128, ReturnWrite)\end{aligned} \tag{22}$$

On applique la règle 2 *Requête de la base de données* avec l'unification

$$[Fid \mapsto \texttt{k\_fdesc}[1]] \text{ car} \tag{23}$$

$$\mathcal{O}^d(D_0) \models^c \texttt{k\_fdesc}[1] = \texttt{k\_fdesc}[1]$$

$\mathbf{P}, \mathbf{D_0} - \ - \vdash max(128, \texttt{k\_fdesc}[1].file.size - \texttt{k\_fdesc}[1].pos, ReqSize)$

$\otimes\ \texttt{k\_mem}[stack] \ll \texttt{k\_fdesc}[1].file$

$\otimes\ \texttt{k\_fdesc}[1].pos := \texttt{k\_fdesc}[1].pos + ReqSize$ (24)

$\otimes\ ReturnRead = ReqSize$

$\otimes\ \texttt{write}(4, \texttt{k\_mem}[stack], 128, ReturnWrite)$

On applique la règle 2 *Requête de la base de données* avec l'unification

$[ReqSize \mapsto 128]$ car

$\mathcal{O}^d(D_0) \models^c \texttt{k\_fdesc}[1].file.size - \texttt{k\_fdesc}[1].pos = 128$

car $\mathcal{O}^d(D_0) \models^c p\_valueInTable(\texttt{k\_fdesc}, 1, pos, 0)$ (25)

$\wedge\ \mathcal{O}^d(D_0) \models^c p\_valueInTable(\texttt{k\_fdesc}, 1, file, \texttt{k\_fs}[fichier1])$

$\wedge\ \mathcal{O}^d(D_0) \models^c p\_valueInTable(\texttt{k\_fs}, fichier1, size, 128)$

$\wedge\ \mathcal{O}^d(D_0) \models^c max(128, 128, 128)$

$\mathbf{P}, \mathbf{D_0} - \ - \vdash \texttt{k\_mem}[stack] \ll \texttt{k\_fdesc}[1].file$

$\otimes\ \texttt{k\_fdesc}[1].pos := \texttt{k\_fdesc}[1].pos + 128$ (26)

$\otimes\ ReturnRead = 128$

$\otimes\ \texttt{write}(4, \texttt{k\_mem}[stack], 128, ReturnWrite)$

On applique la règle 3 *Mise à jour élémentaire de la base de données* avec l'unification vide [] car

$\exists \mathbf{D_1}, \mathcal{O}^t(D_0, D_1) \models p\_add\_flow(\texttt{k\_fs}[fichier1], \texttt{k\_mem}[stack], true)$ car

$\mathcal{O}^d(D_0) \models p\_valueInTable(\texttt{k\_fdesc}, 1, file, \texttt{k\_fs}[fichier1])$

d'après la définition 10, $\mathbf{D_1}$ est tel que

$\mathcal{O}^d(D_1) = \mathcal{O}^d(D_0) \cup p\_flow(1, \texttt{k\_fs}[fichier1], \texttt{k\_mem}[stack], true)$

car $\neg \exists N, Source, Dest, Append,$

$\mathcal{O}^d(D_0) \models p\_flow(N, Source, Dest, Append)$ (27)

$\mathbf{P}, \mathbf{D_1} - \ - \vdash \texttt{k\_fdesc}[1].pos := \texttt{k\_fdesc}[1].pos + 128$

$\otimes\ ReturnRead = 128$ (28)

$\otimes\ \texttt{write}(4, \texttt{k\_mem}[stack], 128, ReturnWrite)$

On applique la règle 3 *Mise à jour élémentaire de la base de données* avec l'unification vide [] car

$\exists \mathbf{D_2}, \mathcal{O}^t(D_1, D_2) \models p\_change(\texttt{k\_fdesc}, 1, pos, 128)$ car

$\mathcal{O}^d(D_1) \models p\_valueInTable(\texttt{k\_fdesc}, 1, pos, 0)$ (29)

$\mathbf{D_2}$ est tel que

$\mathcal{O}^d(D_2) = \mathcal{O}^d(D_1) \cup p\_valueInTable(\texttt{k\_fdesc}, 1, pos, 128)$

$\setminus\ p\_valueInTable(\texttt{k\_fdesc}, 1, pos, 0)$

$$\mathbf{P}, \mathbf{D_2} - \ - \vdash \mathit{ReturnRead} = 128 \otimes \texttt{write}(4, \texttt{k\_mem}[\mathit{stack}], 128, \mathit{ReturnWrite}) \tag{30}$$

On applique la règle 2 *Requête de la base de données* avec l'unification

$$[\mathit{ReturnRead} \mapsto 128] \text{ car } \mathcal{O}^d(D_2) \models 128 = 128 \tag{31}$$

$$\mathbf{P}, \mathbf{D_2} - \ - \vdash \texttt{write}(4, \texttt{k\_mem}[\mathit{stack}], 128, \mathit{ReturnWrite}) \tag{32}$$

On applique la règle 1 *Application de la définition de la transaction.*

$$\mathbf{P}, \mathbf{D_2} - \ - \vdash \texttt{GET\_FID\_FOR\_PROC}(\mathit{procA}, 4, \mathit{Fid}) \otimes \mathit{Fid.opts} \ni \texttt{APPEND} \otimes \mathit{Fid.pos} := \mathit{Fid.file.size} \otimes \mathit{Fid.file} \gg \texttt{k\_mem}[\mathit{stack}] \otimes \mathit{Fid.pos} := \mathit{Fid.pos} + 128 \otimes \mathit{Fid.file.size} := \mathit{Fid.file.size} + 128 \otimes \mathit{ReturnWrite} = 128 \tag{33}$$

On applique la règle 1 *Application de la définition de la transaction.*

$$\mathbf{P}, \mathbf{D_2} - \ - \vdash (\texttt{k\_fdesc}[F].\mathit{procs} \ni \texttt{k\_procs}[\mathit{procA}] \wedge \texttt{k\_fdesc}[F].fd = 4) \otimes \mathit{Fid} = \texttt{k\_fdesc}[F] \otimes \mathit{Fid.opts} \ni \texttt{APPEND} \otimes \mathit{Fid.pos} := \mathit{Fid.file.size} \otimes \mathit{Fid.file} \gg \texttt{k\_mem}[\mathit{stack}] \otimes \mathit{Fid.pos} := \mathit{Fid.pos} + 128 \otimes \mathit{Fid.file.size} := \mathit{Fid.file.size} + 128 \otimes \mathit{ReturnWrite} = 128 \tag{34}$$

On applique la règle 2 *Requête de la base de données* avec l'unification

$$[F \mapsto 2] \text{ car } \mathcal{O}^d(D_2) \models^c \texttt{k\_fdesc}[2].\mathit{procs} \ni \texttt{k\_procs}[\mathit{procA}] \wedge \texttt{k\_fdesc}[2].fd = 4 \tag{35}$$

$$\mathbf{P}, \mathbf{D_2} - \ - \vdash \mathit{Fid} = \texttt{k\_fdesc}[2] \otimes \mathit{Fid.opts} \ni \texttt{APPEND} \otimes \mathit{Fid.pos} := \mathit{Fid.file.size} \otimes \mathit{Fid.file} \gg \texttt{k\_mem}[\mathit{stack}] \otimes \mathit{Fid.pos} := \mathit{Fid.pos} + 128 \otimes \mathit{Fid.file.size} := \mathit{Fid.file.size} + 128 \otimes \mathit{ReturnWrite} = 128 \tag{36}$$

On applique la règle 2 *Requête de la base de données* avec l'unification

$$[\mathit{Fid} \mapsto \texttt{k\_fdesc}[2]] \text{ car } \mathcal{O}^d(D_2) \models^c \texttt{k\_fdesc}[2] = \texttt{k\_fdesc}[2] \tag{37}$$

$$\mathbf{P}, \mathbf{D_2} - - \vdash \texttt{k\_fdesc}[2].opts \ni \texttt{APPEND}$$
$$\otimes\, \texttt{k\_fdesc}[2].pos := \texttt{k\_fdesc}[2].file.size$$
$$\otimes\, \texttt{k\_fdesc}[2].file \gg \texttt{k\_mem}[stack]$$
$$\otimes\, \texttt{k\_fdesc}[2].pos := \texttt{k\_fdesc}[2].pos + 128 \tag{38}$$
$$\otimes\, \texttt{k\_fdesc}[2].file.size := \texttt{k\_fdesc}[2].file.size + 128$$
$$\otimes\, ReturnWrite = 128$$

On applique la règle 2 *Requête de la base de données* avec l'unification vide [] car

$$\mathcal{O}^d(D_2) \models^c p\_valueInTable(\texttt{k\_fdesc}, 2, opts, \{APPEND\}) \tag{39}$$

$$\mathbf{P}, \mathbf{D_2} - - \vdash \texttt{k\_fdesc}[2].pos := \texttt{k\_fdesc}[2].file.size$$
$$\otimes\, \texttt{k\_fdesc}[2].file \gg \texttt{k\_mem}[stack]$$
$$\otimes\, \texttt{k\_fdesc}[2].pos := \texttt{k\_fdesc}[2].pos + 128 \tag{40}$$
$$\otimes\, \texttt{k\_fdesc}[2].file.size := \texttt{k\_fdesc}[2].file.size + 128$$
$$\otimes\, ReturnWrite = 128$$

On applique la règle 3 *Mise à jour élémentaire de la base de données* avec l'unification vide [] car

$$\exists \mathbf{D_3}, \mathcal{O}^t(D_2, D_3) \models p\_change(\texttt{k\_fdesc}, 2, pos, 512) \text{ car}$$
$$\mathcal{O}^d(D_2) \models p\_valueInTable(\texttt{k\_fdesc}, 2, file, \texttt{k\_fs}[fichier2])$$
$$\wedge\, \mathcal{O}^d(D_2) \models p\_valueInTable(\texttt{k\_fs}, fichier2, size, 512) \tag{41}$$
$$\text{d'après 10, } \mathbf{D_3} \text{ est tel que}$$
$$\mathcal{O}^d(D_3) = \mathcal{O}^d(D_2) \cup p\_valueInTable(\texttt{k\_fdesc}, 2, pos, 512)$$
$$\setminus\, p\_valueInTable(\texttt{k\_fdesc}, 2, pos, 0)$$

$$\mathbf{P}, \mathbf{D_3} - - \vdash \texttt{k\_fdesc}[2].file \gg \texttt{k\_mem}[stack]$$
$$\otimes\, \texttt{k\_fdesc}[2].pos := \texttt{k\_fdesc}[2].pos + 128 \tag{42}$$
$$\otimes\, \texttt{k\_fdesc}[2].file.size := \texttt{k\_fdesc}[2].file.size + 128$$
$$\otimes\, ReturnWrite = 128$$

On applique la règle 3 *Mise à jour élémentaire de la base de données* avec l'unification vide [] car

$$\exists \mathbf{D_4}, \mathcal{O}^t(D_3, D_4) \models p\_add\_flow(\texttt{k\_mem}[stack], \texttt{k\_fs}[fichier2], true) \text{ car}$$
$$\mathcal{O}^d(D_4) \models p\_valueInTable(\texttt{k\_fdesc}, 2, file, \texttt{k\_fs}[fichier2])$$
$$\text{d'après la définition 10, } \mathbf{D_4} \text{ est tel que}$$
$$\mathcal{O}^d(D_4) = \mathcal{O}^d(D_3) \cup p\_flow(2, \texttt{k\_mem}[stack], \texttt{k\_fs}[fichier2], true)$$
$$\text{car } \mathcal{O}^d(D_4) \models p\_flow(1, \texttt{k\_fs}[fichier1], \texttt{k\_mem}[stack], true)$$
$$\wedge\, \neg\exists N, Source, Dest, Append,$$
$$\mathcal{O}^d(D_4) \models p\_flow(N, Source, Dest, Append) \wedge N > 1 \tag{43}$$

$\mathbf{P}, \mathbf{D_4} - - \vdash \texttt{k\_fdesc}[2].pos := \texttt{k\_fdesc}[2].pos + 128$

$\otimes\ \texttt{k\_fdesc}[2].file.size := \texttt{k\_fdesc}[2].file.size + 128$ (44)

$\otimes\ ReturnWrite = 128$

On applique la règle 3 *Mise à jour élémentaire de la base de données* avec l'unification vide [] car

$\exists \mathbf{D_5}, \mathcal{O}^t(D_4, D_5) \models p\_change(\texttt{k\_fdesc}, 2, pos, 740)$ car

$\mathcal{O}^d(D_4) \models p\_valueInTable(\texttt{k\_fdesc}, 2, pos, 512) \wedge 512 + 128 = 740$ (45)

d'après la définition 10, $\mathbf{D_5}$ est tel que

$\mathcal{O}^d(D_5) = \mathcal{O}^d(D_4) \cup p\_valueInTable(\texttt{k\_fdesc}, 2, pos, 740)$

$\setminus\ p\_valueInTable(\texttt{k\_fdesc}, 2, pos, 512)$

$\mathbf{P}, \mathbf{D_5} - - \vdash \texttt{k\_fdesc}[2].file.size := \texttt{k\_fdesc}[2].file.size + 128$ (46)

$\otimes\ ReturnWrite = 128$

On applique la règle 3 *Mise à jour élémentaire de la base de données* avec l'unification vide [] car

$\exists \mathbf{D_6}, \mathcal{O}^t(D_5, D_6) \models p\_change(\texttt{k\_fs}, fichier2, size, 740)$ car

$\mathcal{O}^d(D_5) \models p\_valueInTable(\texttt{k\_fdesc}, 2, file, \texttt{k\_fs}[fichier2])$

$\wedge\ \mathcal{O}^d(D_5) \models p\_valueInTable(\texttt{k\_fs}, fichier2, size, 512)$ (47)

$\wedge\ 512 + 128 = 740$

d'après la définition 10, $\mathbf{D_6}$ est tel que

$\mathcal{O}^d(D_6) = \mathcal{O}^d(D_5) \cup p\_valueInTable(\texttt{k\_fs}, fichier2, size, 740)$

$\setminus\ p\_valueInTable(\texttt{k\_fs}, fichier2, size, 512)$

$\mathbf{P}, \mathbf{D_6} - - \vdash ReturnWrite = 128$ (48)

On applique la règle 2 *Requête de la base de données* avec l'unification

$[ReturnWrite \mapsto 128]$ car (49)

$\mathcal{O}^d(D_6) \models 128 = 128$

$\mathbf{P}, \mathbf{D_6} - - \vdash ()$ (50)

qui est vrai par l'axiome 0.

A.3 Trace d'une session $\mathcal{CTR}$

On se place dans le répertoire de l'interpréteur.

```
> swipl
Welcome to SWI-Prolog (Multi-threaded, 64 bits, Version 6.4.1)
% ...

?- [interpctr].
%     library(pairs) compiled into pairs 0,00 sec, 22 clauses
%   library(lists) compiled into lists 0,01 sec, 122 clauses
%   parser compiled into parser 0,01 sec, 165 clauses
```

```
%  compiler compiled into compiler 0,01 sec, 176 clauses
%   syscalls/macros/macros.pl compiled into ctr 0,00 sec, 18 clauses
%  ctr compiled into ctr 0,00 sec, 57 clauses
%  updates compiled into updates 0,00 sec, 10 clauses
%  queries compiled into queries 0,00 sec, 6 clauses
% interpctr compiled into interpctr 0,05 sec, 1,415 clauses
true.

?- ctr_comp('tests/exemple4').
% tests/exemple4.ctr.o compiled into ctr 0,00 sec, 32 clauses
true.

?- load_db_state('tests/exemple4',Procs).
% tests/exemple4.db compiled into interpctr 0,00 sec, 12 clauses
Procs = procs(0, 1) .

?- display.
k_fdesc :
------

k_fs :
------
0 : {type : fichier reg., nom : "source", uid : 0, gid : 0,
    mode : rw-r-----, size : 128}
1 : {type : fichier reg., nom : "tampon", uid : 1, gid : 0,
    mode : rw-rw----, size : 512}

k_mem :
------
0 : {size : 5000, shared : false}
1 : {size : 5000, shared : false}
2 : {size : 3000, shared : false}
3 : {size : 3000, shared : false}
4 : {size : 2000, shared : false}

k_procs :
------
0 : {pid : 3000, ppid : 1, grpid : -1, uid : -1, euid : 1000,
    gid : -1, egid : 1000, sgrps : {},
    sig : [], pending : [], umask : rwxrwxr-x,
    root : 0, wd : -1,
    textMem : 4, stackMem : 0, dataMem : 2, sharedMem : {},
    name : }
1 : {pid : 3001, ppid : 1, grpid : -1, uid : -1, euid : 1001,
    gid : -1, egid : 1001, sgrps : {}, sig : [],
```

```
    pending : [], umask : rwxrwxr-x,
    root : 0, wd : -1,
    textMem : 4, stackMem : 1, dataMem : 3, sharedMem : {},
    name : }

events :
------

true.

?- execc(run(0)),display.
k_fdesc :
------

k_fs :
------
0 : {type : fichier reg., nom : "source", uid : 0, gid : 0,
    mode : rw-r-----, size : 128}
1 : {type : fichier reg., nom : "tampon", uid : 1, gid : 0,
    mode : rw-rw----, size : 56}

k_mem :
------
0 : {size : 5000, shared : false}
1 : {size : 5000, shared : false}
2 : {size : 3000, shared : false}
3 : {size : 3000, shared : false}
4 : {size : 2000, shared : false}
5 : {size : 2000, shared : false}
6 : {size : 5000, shared : false}
7 : {size : 3000, shared : false}
8 : {size : 2000, shared : false}
9 : {size : 5000, shared : false}
10 : {size : 3000, shared : false}

k_procs :
------
0 : {pid : 3000, ppid : 1, grpid : -1, uid : -1, euid : 1000,
    gid : -1, egid : 1000, sgrps : {}, sig : [], pending : [],
    umask : rwxrwxr-x,
    root : 0, wd : -1,
    textMem : 4, stackMem : 0, dataMem : 2, sharedMem : {},
    name : }
1 : {pid : 3001, ppid : 1, grpid : -1, uid : -1, euid : 1001,
    gid : -1, egid : 1001, sgrps : {}, sig : [], pending : [],
```

```
    umask : rwxrwxr-x,
    root : 0, wd : -1,
    textMem : 4, stackMem : 1, dataMem : 3, sharedMem : {},
    name : }
2 : {pid : 3002, ppid : 1, grpid : -1, uid : -1, euid : 1000,
    gid : -1, egid : 1000, sgrps : {}, sig : [], pending : [],
    umask : rwxrwxr-x,
    root : 0, wd : -1,
    textMem : 5, stackMem : 6, dataMem : 7, sharedMem : {},
    name : }
3 : {pid : 3003, ppid : 1, grpid : -1, uid : -1, euid : 1000,
    gid : -1, egid : 1000, sgrps : {}, sig : [], pending : [],
    umask : rwxrwxr-x,
    root : 0, wd : -1,
    textMem : 8, stackMem : 9, dataMem : 10, sharedMem : {},
    name : }

events :
------
0 : processus 0 : open source as file descriptor 0 (global fdesc 0)
1 : processus 0 : open tampon as file descriptor 1 (global fdesc 1)
2 : mem 0 << file 0
3 : processus 0 : read 128 bytes from file descriptor 0 (global fdesc 0)
    into buffer 0
4 : processus 0 : fork to child 2 (text: 5, stack: 6, data: 7)
5 : processus 0 : fork to child 3 (text: 8, stack: 9, data: 10)
6 : mem 0 > file 1
7 : processus 0 : overwrite 128 bytes to file descriptor 1 (global fdesc 1)
from buffer 0
8 : mem 6 > file 1
9 : processus 2 : overwrite 56 bytes to file descriptor 1 (global fdesc 1)
from buffer 6
10 : mem 9 > file 1
11 : processus 3 : overwrite 56 bytes to file descriptor 1 (global fdesc 1)
from buffer 9
12 : processus 0 : close 0
13 : processus 2 : close 0
14 : processus 3 : close 0
15 : processus 0 : close 1
16 : processus 2 : close 1
17 : processus 3 : close 1

true .

?- <Ctrl-D>
```

$\%halt$