LISE
PROJET ANR-07-SESU-007
LIVRABLE D4.1

Titre	Analyse de logs *guidée par des vérifications de formules de responsabilité exprimées en CTL*
Référence	Ce document correspond au livrable D4.1.
Auteurs	Valérie Viet Triem Tong Christophe Bidan, Ludovic Me, Christopher Humphries,Guillaume Piolle
Adresse	prenom.nom@supelec.fr SUPELEC, Equipe SSIR (EA 4039), Rennes
Date	Janvier 2011
Statut	Version finale
Version	1.0

Résumé

Ce document présente une méthode de modélisation d'un ensemble de logs par une structure de Kripke, permettant ainsi son interrogation en utilisant la logique temporelle CTL. Nous prendrons comme exemple une série de logs fournis par l'équipe Amazones de l'Insa de Lyon, que nous modéliserons suivant l'approche proposée. Nous proposons la spécification de quelques propriétés de responsabilité et nous détaillerons l'état actuel du prototype développé.

Table des matières

1 Introduction

Lorsque différents acteurs s'accordent pour réaliser un même service, une même application informatique, ils s'accordent au travers d'un contrat sur les tâches qu'ils devront chacun réaliser. Lors de l'exécution de ce service survient parfois des comportements non prévus au contrat entraînant des dommages pour certaines parties. Les parties lésées peuvent porter l'affaire en justice ou essayer de régler le différend à l'amiable. Dans les deux cas, il faudra savoir mettre en évidence, d'une part, qu'il y a bien eu un dysfonctionnement et d'autre part de savoir attribuer la responsabilité de ce dysfonctionnement à l'une des parties. Le plus souvent, il faudra recourir à un expert qui devra étudier les *traces* qu'ont pu garder les composants informatiques des événements ayant eu lieu afin de mettre en évidence le dysfonctionnement puis d'attribuer l'éventuelle responsabilité de ce dommage. Dans ce travail, nous nous restreignons au cas où les *traces* sont les logs des composants, c'est à dire des documents dans lesquels les composants vont écrire un résumé des actions qu'ils font ou qu'ils observent. Le rôle de l'expert de savoir démêler les informations qu'il trouve dans les logs afin de savoir extraire une séquence d'événements ayant eu lieu et expliquant ou réfutant la plainte de l'un des composants. Dans le travail présenté ici, nous proposons une méthodologie et un outil dédiés à l'analyse de logs permettant d'aider l'expert dans son analyse des logs.

Le logiciel dédié à l'analyse de logs est lui un logiciel critique pour l'ensemble des parties. Nous recommandons que l'analyseur soit développé par un tiers indépendant des parties en présence dans le contrat et qu'il soit développé suivant une méthodologie permettant d'atteindre un haut niveau de confiance. Idéalement nous pensons que les éléments critiques de ce logiciel devraient être certifiés. Outre la automatisation du travail de l'expert, l'analyseur de log offre l'intérêt que es résultats que l'on peut obtenir par lui sont reproductibles : si une partie interroge l'analyseur pour vérifier qu'un ensemble de log permettent de décider qu'un scénario a pu avoir lieu alors une autre partie peut faire la même expérience à un moment différent, les résultats obtenus seront les mêmes.

Dans ce document, nous proposons tout d'abord une modélisation formelle des logs des composants sous la forme d'une structure de Kripke. Nous discuterons ce choix de formalisation dans la section de ce document . Nous proposons dans ce travail de permettre à l'expert d'interroger les logs en utilisant une logique temporelle. Nous avons choisi une logique temporelle car les propriétés de responsabilité intègrent généralement une composante temporelle forte. Nous avons choisi d'utiliser en particulier la logique CTL à cause de la nature des logs étudiés. En effet, CTL se base sur un modèle arborescent du temps : chaque instant dans le temps a éventuellement plusieurs successeurs possibles, sans qu'il soit possible de

distinguer avec certitude lequel surviendra réellement. . Nous montrerons en section 3 de ce document pourquoi cette représentation du temps est une modélisation pertinente.

Le travail présenté ici reprend les log générés par l'équipe AMAZONE dans l'environnement OSGI et présenté dans le livrable LISE D3.1 [5]. Le raisonnement présenté ici étend celui présenté dans l'article [6] dans le sens où nous poursuivons en particulier la réflexion sur l'analyse de log et nous proposons ici de savoir étudier des propriétés intégrant une composante temporelle.

1.1 Cas d'étude

Tout au long de ce document, nous illustrerons nos propos en prenant comme cas d'étude un protocole de signature électronique qui a été étudié dans le cadre du projet LISE. Nous débutons ce document en rappelant ce protocole de signature.

1.2 Description des composants

Notre cas d'étude est un protocole de signature électronique : Un serveur envoie un document à un utilisateur, l'utilisateur peut accepter ou non de signer ce document. Dans le cas où il accepte la signature, il transmet un code PIN à la carte contenue dans son téléphone mobile. La carte signe le document après avoir vérifié la validité du code PIN transmis. Cet exemple est synthétisé par la figure 1.

Cet exemple a été implémenté par l'équipe Amazones de l'Insa de Lyon. Dans l'implémentation, les composants sont simulés par des services web. Chaque service journalise les actions qu'il effectue, les logs étant centralisés. Ce document s'intéresse à l'analyse de ces fichiers de logs pour aider à diagnostiquer les responsabilités en cas de fautes avérées.

2 Réflexion autour de la modélisation des logs

2.1 Terminologie

Dans le travail présenté ici, nous emploierons le terme de log élémentaire ou de ligne de log pour traduire la notion de *log entry* qui exprime une entrée dans un journal de log c'est à dire la transcription d'un évènement observé ou réalisé par un composant. La table 1 présente une ligne de log/ un log élémentaire/ *a log entry* issue du cas d'étude décrit dans la section précédente.

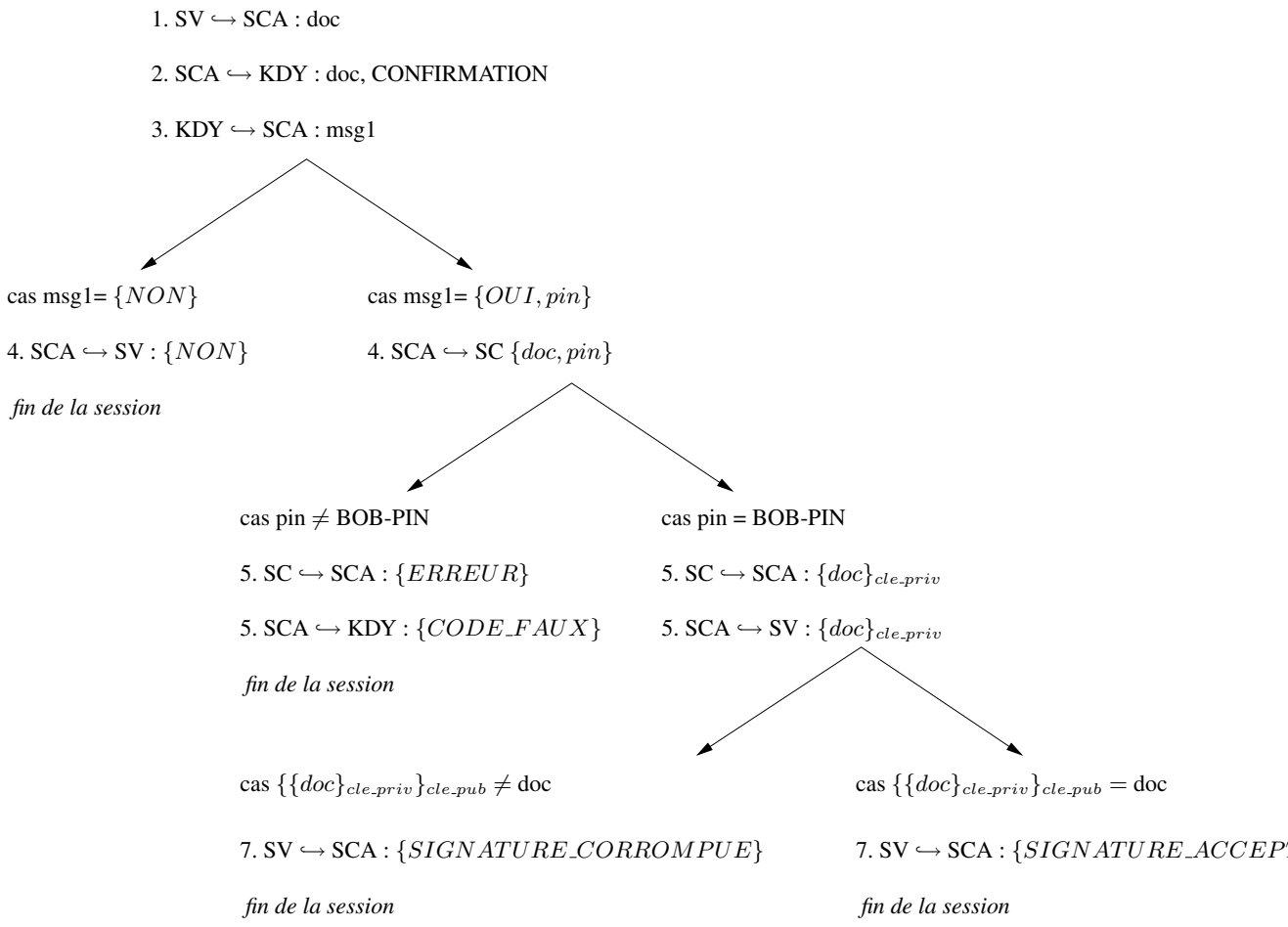


FIGURE 1 – Déroulement de l'exemple 1

Une ligne de log concret :
`c;1;4616686408;2;4;4;3;0;ser_0;`
qui signifie

```
1: Method call at 4616686408:
   uses service
   fr.inria.amazones.lise.app.SigApp
  provided by LISE-SAP-SigApp
  throught interface
  fr.inria.amazones.logos.lisendium.SigAppIfc
   Service user calls function  submitDocToUser
   with folowing parameter(s):
      "To buy this pen at 10\$ enter your PID below."
```

TABLE 1 – Exemple de log issu de notre cas d'étude

Un log élémentaire ou une ligne de log décrit une action effectuée ou observée par un composant. Ce log élémentaire est une vue partielle du système global, celle d'un unique composant à un moment donné.

Ce log élémentaire permet néanmoins de répondre à de premières interrogations. Par exemple, le log précédent permet de savoir que le composant `org.apache.felix.framework` a utilisé le service `fr.inria.amazones.lise.app.SigApp`.

Par abus de langage, nous utiliserons le terme anglais de log pour désigner une collection de log élémentaire, le terme français consacré serait plutôt celui de journal.

Une notion importante pous nous est celle de l'ordre dans une collection de log : peut on toujours définir un ordre total/partiel sur une collection de log ? Nous supposerons que oui dans les sections suivantes et montrerons en section 3.3 comment cette question impacte directement la relation de transition dans la structure de Kripke qui modélise une collection de log.

Nous voyons dès maintenant que l'étude des logs doit nécessairement prendre en compte :

– les informations qu'apporte un log élémentaire ;

– les informations qu'apporte l'enchaînement de ces logs élémentaires.

L'objet de notre travail est de proposer une structure formelle permettant de modéliser ces logs puis de vérifier des propriétés sur cette structure. Nous n'avons pas proposé de structure formelle *ad hoc* mais au contraire, nous avons préféré ré-utiliser des structures logiques bien connues sur lesquelles on sait depuis longtemps faire du model-checking [2], [4], [1].

Dans [8] les auteurs ont exploité une idée similaire : l'objectif du travail présenté dans [8] est de faire de la détection d'intrusion en surveillant les actions faite sur le système via l'analyse d'un fichier de log. La finalité de leur objectif est différente du notre, cependant nous retrouvons l'idée d'étudier les actions faites par/sur un système par l'analyse de fichier de log. Dans [8], les auteurs utilisent des formules de logique temporelle pour encoder des scénario d'attaque. Ces formules sont ensuite vérifiées à la volée dans les logs du système afin de détecter des intrusions contre ce système. Comme dans [8], nous pensons que les logiques temporelles offrent un bon moyen de spécifier puis vérifier des propriétés sur des logs.

Concernant les logs eux-même, nous pensons que les structures des Kripke sont des objets formels qui permettent de bien les représenter et les manipuler. Pour modéliser les logs nous utiliserons la correspondance décrite informellemnt dans la figure 2.1. Dans la suite de ce travail, nous motivons cette proposition en montrant d'abord pourquoi cette représentation nous semble naturelle (dans les sections 3.2, 3.3 et 3.4). Nous montrerons ensuite que l'intérêt majeur de cette représentation est de permettre l'utilisation de la logique CTL et ainsi la vérification du fait qu'un log exhibe une propriété ou pas. La dernière partie de ce travail consistera donc à exprimer

1 log élémentaire l_{elem}	$\leftrightsquigarrow$	1 état s_l
1 log (collection finie de logs élémentaires)	$\leftrightsquigarrow$	un ensemble fini d'états
ordonnancement des logs	$\leftrightsquigarrow$	relation de transition entre les états
informations apprises dans un log élémentaire l_{elem}	$\leftrightsquigarrow$	ensemble des prédicats vrais dans s_l

FIGURE 2 – Correspondance intuitive entre logs et structures de Kripke

des propriétés de responsabilité en utilisant cette logique.

3 Modélisation par des structures de Kripke

3.1 Rappels sur les structures de Kripke

Une structure de Kripke est une machine du type état-transition, où chaque état est étiqueté par l'ensemble des prédicats vrais sur cet état. Dans une structure de Kripke, la relation de transition entre les états permet de passer d'un état à un autre et lorsque cette relation est sérielle, chaque état est en relation avec au moins un état (éventuellement le même). Cette propriété est importante car dans le cas d'une relation de transition sérielle, les chaînes d'états de la structure correspondante sont nécessairement infinies.

Formellement, une structure de Kripke $\mathcal{K}$ est la donnée de $(\mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L})$

- $\mathcal{S}$ un ensemble fini d'états ;
- $\mathcal{I} \subseteq \mathcal{S}$ un ensemble d'états initiaux ;
- $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$ une relation de transition telle que pour chaque état s dans $\mathcal{S}$ il existe au moins un s' dans $\mathcal{S}$ vérifiant $s\mathcal{R}s'$;
- $\mathcal{A}$ un ensemble de propositions atomiques ;
- $\mathcal{L} : \mathcal{S} \to 2^{\mathcal{A}}$ une fonction d'étiquetage sur les états telle que $\mathcal{L}(s)$ est l'ensemble des propositions atomiques valides dans l'état s.

Une exécution dans une telle structure est un élément de $\mathcal{S}^\star$, c'est à dire une séquence d'états $\pi = s_0 s_1 s_2 s_3 s_4 \ldots$ éventuellement infinie de $\mathcal{S}$ tel que s_0 est un état initial et que les états de la séquence sont liés par la relation de transition $\mathcal{R} : \forall s_i \in \pi,\ s_i \mathcal{R} s_{i+1}$. L'ensemble des chemins d'exécution possibles est appelé arbre des exécutions ou dépliage de la structure de Kripke.

Cette définition, générale, permet éventuellement de représenter dans la même structure de Kripke plusieurs arborescences, donc plusieurs fichiers

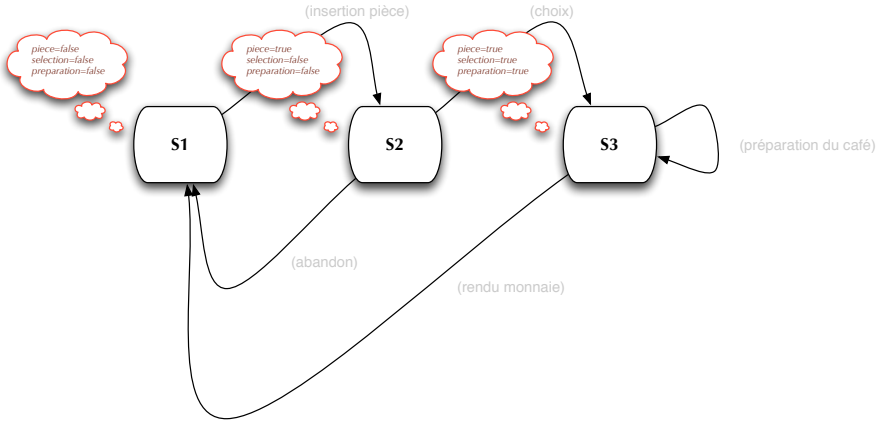


FIGURE 3 – Une structure de Kripke représentant une machine à café

de logs appartenant à différents composants d'un même système. Dans l'état actuel de nos travaux, nous nous restreindrons à un état initial unique et donc à une arborescence unique, représentant un fichier de log unique.

La figure 3.1 est un un exemple très simple de structure de Kripke qui représente une machine à café. Cette structure est définie par :

- $\mathcal{S} = \{s_1, s_2, s_3\}$ l'ensemble des d'états
- $\mathcal{I} = \{s_1\}$ l'ensemble des états initiaux
- $\mathcal{R} = \{(s_1, s_2), (s_2, s_3), (s_3, s_3), (s_3, s_1)\}$ la relation de transition (cette relation est sérielle, et en l'occurence totale).
- $\mathcal{A} = \{piece, selection, preparation\}$
- $\mathcal{L}(s_1) = \emptyset, L(s_2) = \{piece\}, L(s_3) = \{piece, selection, preparation\}$

Les différentes étapes de la construction de notre modélisation des logs par une structure de Kripke sont la définition de l'ensemble des états, de l'ensemble des propositions atomiques, de la fonction d'étiquetage des états et enfin de la définition de la relation de transition. Nous étudions dans la suite la définition de l'ensemble des propositions atomiques puis nous proposerons une définition constructive de l'ensemble des états conjointement à la fonction d'étiquetage des états.

3.2 Représentation d'un log élémentaire par un état

3.2.1 Fonction d'étiquetage

Nous proposons d'associer un état à chaque log élémentaire. Pour cela il nous faut tout d'abord définir quelles sont les propriétés qui nous intéressent sur les logs. En effet, ces propriétés permettront de définir l'ensemble des

propositions atomiques qui nous intéressent et, en conséquence, la fonction d'étiquetage. Reprenons l'exemple de notre cas d'étude en particulier le log élémentaire présenté sur la table 1. D'un log de ce type nous pouvons apprendre :

- si le log est relatif à une demande de service ou une réponse ;
- le nom du composant appelant (il y a 5 composants possibles dans notre étude) ;
- l'heure de l'appel (aspect non encore traité en l'état actuel du prototype) ;
- le nom du service appelé ;
- le nom du fournisseur de service ;
- le nom de l'interface utilisée ;
- le nom de la fonction appelée ainsi que les paramètres utilisés.

Nous proposons d'associer une proposition atomique à chaque information pertinente. L'ensemble de ces propositions atomiques peut simplement être l'ensemble des composants, services, fournisseurs, interfaces, fonctions ... Cette définition doit se faire directement en lien avec l'expression des propriétés jugées utiles pour la responsabilité. Dans la suite de ce document, nous détaillons un court exemple issu du cas d'étude du projet LISE dans lequel nous détaillons l'ensemble des propositions atomiques utilisé pour décrire les logs. Nous renvoyons donc notre lecteur à la section 5.2 pour une spécification détaillée des variables propositionnelles.

3.2.2 Définition des états de la structure

Chaque log élémentaire est associé à un état de la structure, cet état est étiqueté par les propositions atomiques vraies dans cet état.

3.3 Ordonnancement des logs et impacts sur la relation de transition

Dans une structure de Kripke, la relation de transition est une relation entre les états, c'est un élément de $\mathcal{S} \times \mathcal{S}$. Dans ce travail, nous interprétons cette relation entre les états comme une relation temporelle :
Pour deux états s et s', si

$s\mathcal{R}s'$ alors ici que l'événement que modélise s' a lieu après l'événement que modélise s (voir illustration figure 4, cas 1)

$s\mathcal{R}s'$**,** $s\mathcal{R}s''$ **et** $\neg(s'\mathcal{R}s'')$**,** $\neg(s''\mathcal{R}s')$ exprime que l'événement que modélise s' a lieu après l'événement que modélise s, l'événement que modélise s" a lieu après l'événement que modélise s mais qu'on ne peut pas donner de relation temporelle entre s' et s" (voir illustration figure 4, cas 2). Il y a donc une incertitude dans l'ordre

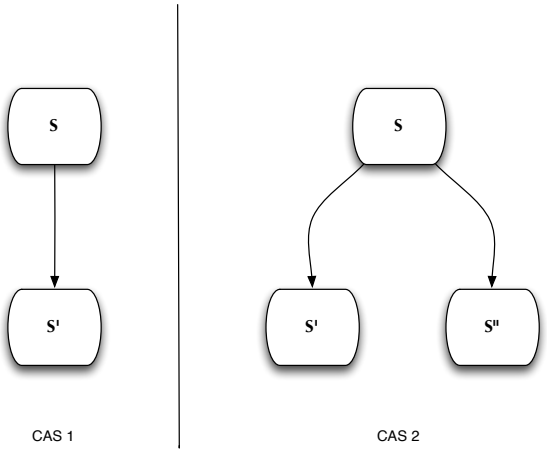


FIGURE 4 – Relation de transition entre les état, relation avec l'ordonnancement des logs

des évènements s' et s" dont on sait par ailleurs qu'ils ont eu lieu après s.

Pour retracer les événements susceptibles d'avoir eu lieu en fonction de ce que l'on peut observer dans les logs, nous devons reconstituer l'ordre chronologique dans lequel ces événements ont eu lieu. Dans l'architecture physique collectant les logs, il y a une première relation temporelle naturelle qui est l'ordre d'apparition dans le fichier de log : si un évènement (ou log élémentaire) A est inscrit avant un autre évènement B on peut se persuader que A est observé avant B. Si A et B se suivent dans les logs, il parait naturel que cela se reflète dans la relation de transition et donc si s_A (resp. s_B) est l'état modélisant le log élémentaire A (resp. B) nous avons $s_A \mathcal{R} s_B$. Cette réflexion n'est valable que dans l'hypothèse où l'on peut assurer que l'ordre du déroulement des évènements correspond à l'ordre d'apparition dans les logs. Par exemple, dans un fichier de log ne journalisant que les actions faites par un seul composant, nous pourrons accepter l'hypothèse que l'ordre d'apparition des événements correspond à l'ordre d'inscription des logs élémentaires.

Par contre, si la collection de log contient les journaux de plusieurs composants, ces composants ont un accès concurrent à ce fichier. Chaque composant bloque l'accès au fichier lorsqu'il journalise ses actions, ce qui peut retarder un autre composant qui journalisera alors ses propres événements avec retard. L'ordre d'inscription des événements dans le log peut alors être différents de l'ordre apparition réels des événements.

Dans l'exemple que nous avons considéré, les composants sont simulés par une orchestration de web service et les logs sont centralisé. Nous pouvons donc considérer que la relation d'ordre entre les évènements corres-

pond à l'ordre d'apparition dans le journal de logs. Du point de vue de la modélisation par une structure de Kripke cela signifie que sans cette structure chaque état modélisant un événement est en relation avec au plus un autre état. Néanmoins, de façon générale la modélisation que nous proposons permet de prendre en compte des cas où l'on ne dispose que d'un ordre partiel sur les états et notre outil implémente ce cas général. Il faut cependant retenir que la définition de la relation de transition de la structure de Kripke est un travail délicat, qu'il impacte directement les résultats que fournira l'analyseur. Cette relation de transition est à construire suivant l'architecture des composants et l'architecture de log.

3.4 Propriétés sur les états et sur les chemins

La modélisation des logs par des structures de Kripke permet de distinguer, parmi les propriétés sur les logs élémentaires, celles qui sont des propriétes sur les états de la structure et qui sont représentés à l'aide de l'étiquette de chaque état, de celles qui concernent la séquence d'exécution des évènements représentés et qui sont ici transcrits par la relation de transition entre les états. Il paraît dès lors naturel d'utiliser une logique comme CTL pour interroger la modélisation des logs sous forme de structure de Kripke.

4 Utilisation de CTL

4.1 Pourquoi CTL ?

La logique qui est choisie ici pour spécifier les propriétés de responsabilité sur les logs est la logique temporelle CTL. Nous avons choisi une logique temporelle car nous pensons que la notion d'ordre dans la séquence des événements apporte un niveau de détails meilleur que celui qui avait été utilisé dans le travail précédent du projet [6]. Nous avons choisi CTL plutot que LTL car le modèle sous-jacent du temps dans CTL est arborescent, ce qui permet en particulier le modéliser l'incertitude entre l'ordre d'apparition des événements. Ceci est détaillé en section 3.3

4.2 Présentation de CTL

La logique CTL, ou logique du temps arborescent (*Computational Tree Logic*), a été définie par Clarke et Emerson en 1981. C'est une logique modale proposant des modalités temporelles. Le modèle du temps considéré dans CTL est dit *arborescent* dans le sens où un état a éventuellement plusieurs successeurs possibles.

Les formules de CTL combinent des opérateurs sur le temps et des quantificateurs sur les chemins. Ce sont des formules de la forme $QO\Phi$. Q est un quantificateur sur les chemins qui vaut soit A (quantificateur sur les chemins signifiant *pour tous les chemins*) soit E (quantificateur sur les chemins signifiant *il existe un chemin*). O est un opérateur modal temporel, nous considérerons ici que O est pris parmi $\{X, \Box, \Diamond\}$. Plus précisément, les formules de CTL sont définies récursivement à partir des « formules d'état » et des « formules de chemin » définies comme suit :

- **Formules d'état**

  True, False sont des formules d'états ;

  les variables propositionnelles sont des formules d'états ;

  $\neg\Phi$ **et** $\Phi \vee \Psi$ (permettant de construire les opérateurs $\wedge$ et $\rightarrow$) sont des formules d'état si Φ et Ψ sont des formules d'état ;

  $\mathbf{A}\Phi$ est une formule d'état si Φ est une formule de chemin (intuitivement, AΦ signifie Φ *est satisfaite dans toutes les exécutions possibles à partir de l'état courant*) ;

  $\mathbf{E}\Phi$ est une formule d'état si Φ est une formule de chemin (intuitivement, EΦ signifie Φ *est satisfaite pour au moins une exécution possible à partir de l'état courant*).

- **Formules de chemin :**

  une formule d'état est une formule de chemin ;

  $\neg\Phi$ **et** $\Phi \vee \Psi$ sont des formules de chemin si Φ et Ψ sont des formules de chemin ;

  $X\Phi$ `(NEXT)` est une formule de chemin si Φ est une formule d'état ;

  $\Diamond\Phi$ est une formule de chemin si Φ est une formule d'état (intuitivement, $\Diamond\Phi$ signifie Φ *sera vraie un jour* ;

  $\Box\Phi$ est une formule de chemin si Φ est une formule d'état (intuitivement, $\Box\Phi$ signifie Φ *sera toujours vraie dans le futur*.

CTL est ensuite défini comme le sous-ensemble des formules d'état dans lesquelles X, $\Diamond$ et $\Box$ sont toujours précédés d'un opérateur A ou E et réciproquement.

4.3 Evaluation sur les traces infinies

On peut interpréter une formule Φ de CTL sur la structure $\mathcal{K}$ et ainsi répondre à la question : *Est-ce que la propriété décrite dans la formule* Φ *est satisfaite par la structure* $\mathcal{K}$ *?* ou en d'autres termes, *est-ce que la structure est un modèle de la formule ?* La logique CTL s'interprète sur des structures de Kripke : on dit qu'une structure $\mathcal{K}$ satisfait une formule Φ et on note $\mathcal{K} \models \Phi$ si l'état initial de $\mathcal{K}$ satisfait Φ. Dans une structure de Kripke, la

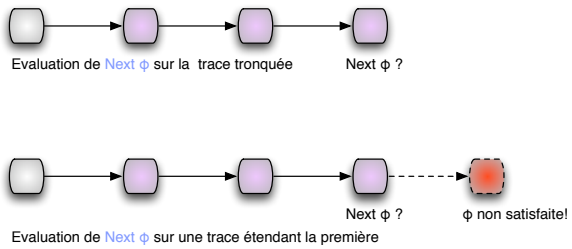


FIGURE 5 – Problème de l'évaluation de Next sur une trace tronquée

trace des évènements correspondant au dépliage de la structure, ce dépliage est un arbre infini dès que la relation de transition est sérielle.

4.4 Evaluation sur des traces finies

Dans le cas de l'analyse de logs, nous n'aurons qu'une vue partielle du système : celle de l'ensemble des exécutions ayant eu lieu dans une période de temps finie. La définition de cette période de temps est un problème important qui sort du cadre de l'analyse à proprement parler et qui devrait être (à notre avis) définie dans le cadre du contrat qui lie les composants en présence. De façon générale, une trace d'exécution peut être infinie, maximale ou tronquée. Une trace est infinie lorsque chaque état possède au moins un état suivant, une trace est maximale lorsque qu'elle n'est pas un préfixe strict d'une trace plus longue et elle est tronquée lorsqu'elle est préfixe strict d'une autre trace plus longue. Une portion de log est typiquement une trace tronquée dans le sens où c'est une trace qui peut être prolongée en considérant une collection de logs qui l'englobent.

Si CTL est une logique qui s'interprète sur des traces infinies, il faut adapter le raisonnement aux traces finies et au cas particuliers des traces tronquées. Ce problème a été abordé dans [3] par Eisner *et al.* pour la logique temporelle LTL. Le fait que les traces soient finies pose un problème d'interprétation des formules sur le dernier état de la trace, en particulier celles des formules de chemins commençant par l'opérateur `Next`. Intuitivement, la question qui se pose est de savoir comment évaluer sur un état s une formule de la forme $X\Phi$ (signifiant $\ll \Phi$ est satisfaite dans l'état suivant $s \gg$) quand s est le dernier état de la séquence t, donc n'a pas de suivant sur la trace t ? Plus encore, si la trace étudiée est une trace tronquée, cela signifie qu'il existe peut-être une trace t' telle que t est préfixe de t' et dans laquelle il est possible d'évaluer Φ. Le dessin de la figure 4.4 illustre ce problème.

En terme d'implémentation, nous choisirons de lancer une exception lorsque nous arriverons dans ce cas de figure. Ce choix d'implémentation pourra être revu dans l'avancement du projet LISE.

4.5 Sémantique de CTL

Considérons $\mathcal{K}$ est une structure de Kripke considérée via son dépliage, c'est à dire l'ensemble des exécutions possibles sur $\mathcal{K}$.

On peut interpréter une formule Φ de CTL sur la structure $\mathcal{K}$ et ainsi répondre à la question : *Est-ce que la propriété décrite dans la formule* Φ *est satisfaite par la structure* $\mathcal{K}$ *?* ou en d'autres termes, *est-ce que la structure est un modèle de la formule ?*

On dira qu'une structure de Kripke $\mathcal{K}$ satisfait une formule CTL Φ (noté $\mathcal{K} \models \Phi$) si et seulement si l'état initial s_0 satisfait Φ (noté $\mathcal{K}, s_0 \models \Phi$). Pour un chemin π d'origine s_0, $\pi, s \models \Phi$ signifie que Φ (qui doit être une variable de chemin) est vérifiée à l'état s dans le chemin et $\mathcal{K}, \pi \models \Phi$, équivalent à $\pi, s_0 \models \Phi$, que Φ est vérifiée à l'origine du chemin (on dira alors que le chemin vérifie Φ). La relation de satisfaction pour un état quelconque du dépliage, notée $\mathcal{K}, s \models \Phi$, est définie par induction structurelle de la façon suivante :

1. **Formules d'états**

   True, False
   $\mathcal{K}, s \models true$ et $\mathcal{K}, s \not\models false$ pour toute structure $\mathcal{K}$ et tout état s dans son dépliage.

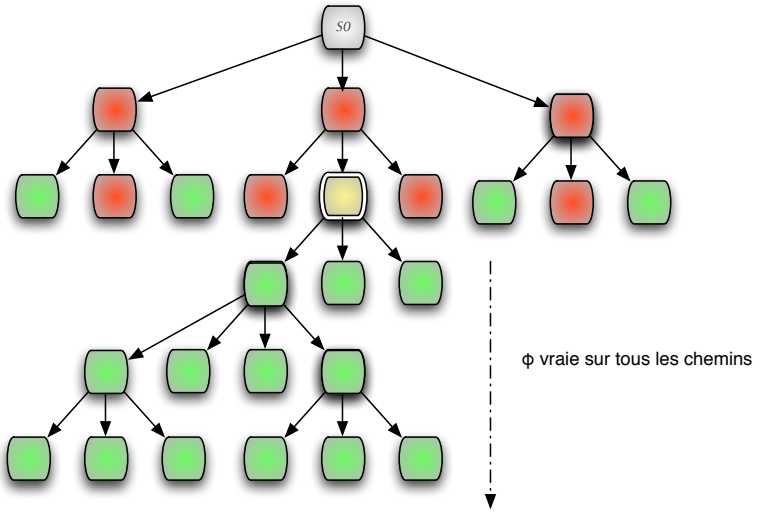
   les variables propositionnelles
   $\mathcal{K}, s \models P$ si et seulement si $P \in \mathcal{L}(s)$

   $\mathcal{K}, s \models \neg\Phi$ si et seulement si $\mathcal{K}, s \not\models \Phi$

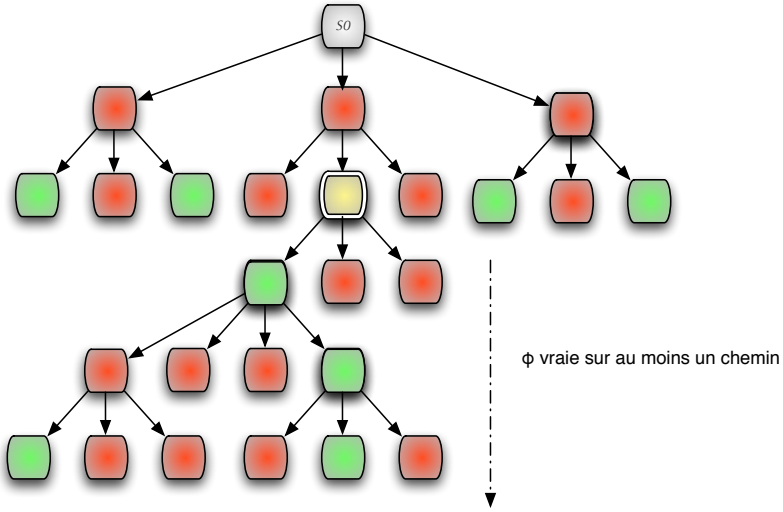
   $\mathcal{K}, s \models \Phi \vee \Psi$ si et seulement si $\mathcal{K}, s \models \Phi$ ou $\mathcal{K}, s \models \Psi$

   $\mathcal{K}, s \models \Phi \wedge \Psi$ si et seulement si $\mathcal{K}, s \models \Phi$ et $\mathcal{K}, s \models \Psi$

   $\mathcal{K}, s \models A\Phi$ si et seulement si pour tout chemin π d'origine s dans $\mathcal{K}$ on a $\pi \models \Phi$ (dans ce cas, suivant la syntaxe définie précédemment, Φ est une formule de chemin).



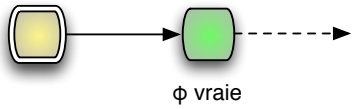
$\mathcal{K}, s \models E\Phi$ si et seulement si il existe un chemin π d'origine s dans $\mathcal{K}$ tel que $\pi \models \Phi$ (dans ce cas, suivant la syntaxe définie précédemment, Φ est une formule de chemin).



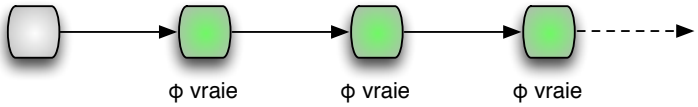
2. **Formules de chemin** Un chemin dans une structure de Kripke une séquence d'état $\pi = s_0 s_1 s_2 ..$, tel que $\forall \geq 0, s_i \mathcal{R} s_{i+1}$. La satisfaction d'une formule Φ par l'état s_i d'une séquence π, notée $\pi, s_i \models \Phi$, est définie de la manière suivante :

   $\pi, s_i \models \Phi$ (où Φ est une formule d'état) si et seulement si $\mathcal{K}, s_i$;

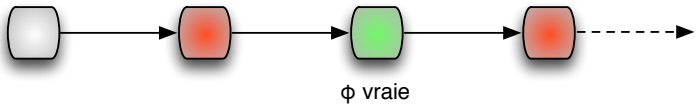
   $\pi, s_i \models \mathbf{X}\Phi$ si et seulement si $\pi, s_{i+1} \models \Phi$ (en d'autres termes, Φ est vérifié par l'état successeur dans le chemin) ;



$\pi, s_i \models \Box\Phi$ si et seulement si $\forall j \geq i, \pi, s_j \models \Phi$ (en d'autres termes, Φ est vraie à l'état s_i et le demeure pour tout le reste du chemin) ;



$\pi, s_i \models \Diamond\Phi$ si et seulement si $\exists j \geq i, \pi, s_j \models \Phi$ (en d'autres termes, soit Φ est vraie à l'état s_i, soit elle le deviendra à un état futur sur le chemin).



5 Modélisation d'un extrait du cas d'étude

On considère un collection de logs élémentaires. Cette collection est donnée sous forme d'un texte lisible. En pratique, un log élémentaire est plutot représenté par une liste d'entier ou de chaîne de caractère du type de celle donné en exemple en section 2.

```
1: Method call at 4616686408:
   org.apache.felix.framework
   uses service fr.inria.amazones.lise.app.SigApp
   provided by LISE-SAP-SigApp
   throught interface fr.inria.amazones.logos.lisendium.SigAppIfc
   Service user calls
   function boolean submitDocToUser(java.lang.String)
    with folowing parameter(s):
      "To buy this pen at 10$ enter your PID below."

2: Method call at 4620299367:
   LISE-SAP-SigApp
   uses service fr.inria.amazones.lise.disp.MobileIOImpl
    provided by LISE-MPP-MobileIO
    throught interface fr.inria.amazones.logos.lisendium.MobileIOIfc
   Service user calls
   function java.lang.String submitDocToUser(java.lang.String)
   with folowing parameter(s):
      "To buy this pen at 10$ enter your PID below."
```

```
2’ : Method respond at 8532827286:
      "123"

3: Method call at 8533162524:
   LISE-SAP-SigApp
   uses service fr.inria.amazones.lise.card.CardImpl
   provided by LISE-SCP-Card
   throught interface fr.inria.amazones.logos.lisendium.CardIfc
   Service user calls
   function java.lang.String sign(java.lang.String,int)
   with folowing parameter(s):
      "To buy this pen at 10$ enter your PID below."
      123

3’ : Method respond at 8533302626:
      ""

4: Method call at 8533425337:
   LISE-SAP-SigApp
   uses service fr.inria.amazones.lise.disp.MobileIOImpl
   provided by LISE-MPP-MobileIO
   throught interface fr.inria.amazones.logos.lisendium.MobileIOIfc
   Service user calls
   function java.lang.String submitDocToUser(java.lang.String)
   with following parameter(s):
      "To buy this pen at 10$ enter your PID below."

4’ : Method respond at 12738414453:
      "1234"

5: Method call at 12738629983:
   LISE-SAP-SigApp
   uses service fr.inria.amazones.lise.card.CardImpl
   provided by LISE-SCP-Card
   throught interface fr.inria.amazones.logos.lisendium.CardIfc
   Service user calls
   function java.lang.String sign(java.lang.String,int)
   with following parameter(s):
      "To buy this pen at 10$ enter your PID below."
      1234

5’ : Method respond at 12738844256:
```

```
      "To buy this pen at 10$ enter your PID below.
      (with private key="privateKey" and public key = "publicKey")"

1' : Method respond at 12738963196:
      true

6: Method call at 15010081589:
   org.apache.felix.framework
   uses service fr.inria.amazones.lise.app.SigApp
   provided by LISE-SAP-SigApp
   throught interface fr.inria.amazones.logos.lisendium.SigAppIfc
   Service user
   calls function boolean submitDocToUser(java.lang.String)
   with following parameter(s):
      "To buy this pen at 10$ enter your PID below."

7: Method call at 15010270301:
   LISE-SAP-SigApp
   uses service fr.inria.amazones.lise.disp.MobileIOImpl
   provided by LISE-MPP-MobileIO throught
   interface fr.inria.amazones.logos.lisendium.MobileIOIfc
   Service user
   calls function java.lang.String submitDocToUser(java.lang.String)
   with folowing parameter(s):
      "To buy this pen at 10$ enter your PID below."

7' : Method respond at 21016999996:
      "azer"

8: Method call at 21019733444:
   LISE-SAP-SigApp
   uses service fr.inria.amazones.lise.disp.MobileIOImpl
   provided by LISE-MPP-MobileIO
   throught interface fr.inria.amazones.logos.lisendium.MobileIOIfc
   Service user
   calls function java.lang.String submitDocToUser(java.lang.String)
    with folowing parameter(s):
      "To buy this pen at 10$ enter your PID below."

8' : Method respond at 28524514917:
      "zerty"

9: Method call at 28524783178:
```

```
   LISE-SAP-SigApp
   uses service fr.inria.amazones.lise.disp.MobileIOImpl
   provided by LISE-MPP-MobileIO
   throught interface fr.inria.amazones.logos.lisendium.MobileIOIfc
   Service user
   calls function java.lang.String submitDocToUser(java.lang.String)
   with folowing parameter(s):
      "To buy this pen at 10$ enter your PID below."

9' : Method respond at 32128165730:
      "erty"

6' : Method respond at 32128425680:
      false
```

5.1 Construction de la structure de Kripke représentant la collection de logs

5.1.1 Ensemble des états

Nous associons un état à chaque log élémentaire. Dans l'exemple proposé il y a 18 logs élémentaires donc 18 états, parmi lequel l'état initial s_0 correspond au premier log apparaissant dans la liste.

Nous avons donc défini l'ensemble des états $\mathcal{S} = \{s_0, s_1, \ldots, s_{17}\}$ et l'ensemble des états initiaux $\mathcal{I} = \{s_0\}$

5.2 Ensemble des variables propositionnelles

Appels / Retour de services
Dans cette collection de logs élémentaires, nous pouvons distinguer les logs correspondant à des appels des logs et des logs correspondant à des retours de service. Nous définissons donc deux propositions atomiques `Call` et `Response`. `Call` (resp. `Response`) vaut vrai si et seulement si le log désigne un appel de service (resp. retour de service).

Composants appelants
Lorsqu'un log correspond à un appel, il contient le nom du composant appelant. Dans cet exemple, il y a 2 composants possibles. Nous définissons donc 2 propositions atomiques de la forme `Caller_x` qui sont vraies si et seulement si le composant x est le composant appelant. La valeur de `x` est à choisir dans l'ensemble

$$\{\texttt{org.apache.felix.framework}, \texttt{LISE-SAP-SigApp}\ \}$$

Service utilisé
Lorsqu'un log correspond à un appel, il contient aussi le nom du service utilisé. Dans cet exemple, il y a 3 services possibles. Nous définissons donc 3 propositions atomiques de la forme `Service_x` qui sont vraies si et seulement si le service x est le service utilisé. La valeur de `x` est à choisir dans l'ensemble

{`fr.inria.amazones.lise.app.SigApp`,
`fr.inria.amazones.lise.disp.MobileIOImpl`,
`fr.inria.amazones.lise.card.CardImpl`}

Fournisseur de services
Lorsqu'un log correspond à un appel, il contient le nom du fournisseur de service. Dans ce petit exemple, il y a 3 fournisseurs possibles. Nous définissons donc 3 propositions atomiques de la forme `Provider_x` qui sont vraies si et seulement si x est le fournisseur de service. La valeur de `x` est à choisir dans l'ensemble

{ `LISE-SAP-SigApp`,
`LISE-MPP-MobileIO` ,
`LISE-SCP-Card` }

Interface utilisée
Lorsqu'un log correspond à un appel, il contient le nom de l'interface utilisée. Dans cet exemple, il y a 2 interfaces possibles. Nous définissons donc 2 propositions atomiques de la forme `Interface_x` qui sont vraies si et seulement si l'interface x est l'interface utilisée. La valeur de `x` est à choisir dans l'ensemble

{`fr.inria.amazones.logos.lisendium.SigAppIfc`,
`fr.inria.amazones.logos.lisendium.MobileIOIfc`}

Fonction appelée
Lorsqu'un log correspond à un appel, il contient le nom de la fonction appelée. Dans cet exemple, il y a 2 fonctions possibles. Nous définissons donc 2 propositions atomiques de la forme `Function_x` qui sont vraies si et seulement si la fonction x est la fonction appelée. La valeur de `x` est à choisir dans l'ensemble

{`submitDocToUser`,
`sign`}

Paramètres
Les paramètres passés aux fonctions appelées ne sont pas nécessairement journalisés. Par exemple, le code pin d'un utilisateur ne devrait pas figurer dans les logs, sinon les logs offriraient une manière simple d'apprendre le code d'un utilisateur. Pire encore, les paramètres peuvent prendre des valeurs absolument arbitraires puisque dans le cas général, ils peuvent sortir de l'imagination d'un utilisateur. Nous proposons qu'un paramètre soit

simplement représenté par une proposition `param_x` qui soit une chaîne de caractères.

La table 5.2 synthétise l'ensemble des propositions atomiques définies dans cette section.

5.2.1 Fonction d'étiquetage

La fonction d'étiquetage $\mathcal{L}$ associe à chaque état de la structure l'ensemble des propositions atomiques vraies dans cet état. Pour notre cas d'étude, $\mathcal{L}$ est définie par :

- $\mathcal{L}(s_0) = \{$`Caller_org.apache.felix.framework(1)`,
  `Service_fr.inria.amazones.lise.app.SigApp`,
  `Provider_LISE-SAP-SigApp`,
  `Interface_fr.inria.amazones.logos.lisendium.SigAppIfc`,
  `Function_submitDocToUser`,
  `"To buy this pen at 10$ enter your PID below."`$\}$

- $\mathcal{L}(s_1) = \{$`Caller_LISE-SAP-SigApp(2)`,
  `Service_fr.inria.amazones.lise.disp.MobileIOImpl`,
  `Provider_LISE-MPP-MobileIO` ,
  `Interface_fr.inria.amazones.logos.lisendium.MobileIOIfc`,
  `Function_submitDocToUser`,
  `"To buy this pen at 10$ enter your PID below."`$\}$

- $\mathcal{L}(s_2) = \{$`Response(2)`,`"123"`$\}$

- $\mathcal{L}(s_3) = \{$`Caller_LISE-SAP-SigApp(3)`,
  `Service_fr.inria.amazones.lise.card.CardImpl` ,
  `Provider_ LISE-SCP-Card`,
  `Interface_ fr.inria.amazones.logos.lisendium.CardIfc`,
  `Function_sign`,
  `"To buy this pen at 10$ enter your PID below.", "123"`$\}$

- $\mathcal{L}(s_4) = \{$`Response(3)`,`""`$\}$

- $\mathcal{L}(s_5) = \{$`Caller_LISE-SAP-SigApp(4)`,
  `Service_ fr.inria.amazones.lise.disp.MobileIOImpl` ,
  `Provider_ LISE-MPP-MobileIO`,
  `Interface_fr.inria.amazones.logos.lisendium.MobileIOIfc`,
  `Function_submitDocToUser`,

Proposition Atomiques	Signification
Appels/Retour	
`Call`	Ce log désigne un appel de service
`Response`	Ce log désigne un retour de service
Composants appelant	
`Caller_x`	le composant appelant est `x` où `x` ∈ {`org.apache.felix.framework`, `LISE-SAP-SigApp` }
Service utilisé	
`Service_x`	le service utilisé est `x` où `x` ∈ {`fr.inria.amazones.lise.app.SigApp`, `fr.inria.amazones.lise.disp.MobileIOImpl`, `fr.inria.amazones.lise.card.CardImpl`,}.
Fournisseur	
`Provider_x`	le fournisseur de service est `x` où `x` ∈ {`LISE-SAP-SigApp`, `LISE-MPP-MobileIO` , `LISE-SCP-Card` ,}.
Interface	
`Interface_x`	l'interface de service est `x` où `x` ∈ {`fr.inria.amazones.logos.lisendium.SigAppIfc`, `fr.inria.amazones.logos.lisendium.MobileIOIfc`,}.
Fonction	
`Function_x`	la fonction utilisée est `x` où `x` ∈ {`submitDocToUser`, `sign`}.
Paramètres	
`param_x`	les paramètres utilisés sont égaux à la chaîne de caractère `param_x`.

TABLE 2 – Synthèse des propositions atomiques utilisées

"To buy this pen at 10$ enter your PID below."}

- $\mathcal{L}(s_6) = \{$Response(4),"1234"$\}$

- $\mathcal{L}(s_7) = \{$Caller_LISE-SAP-SigApp(5),
  Service_fr.inria.amazones.lise.card.CardImpl ,
  Provider_ LISE-SCP-Card,
  Interface_ fr.inria.amazones.logos.lisendium.CardIfc,
  Function_sign,
  "To buy this pen at 10$ enter your PID below.", "1234"$\}$

- $\mathcal{L}(s_8) = \{$Response(5),
  "To buy this pen at 10$ enter your PID below.
  (with private key="privateKey" and public key = "publicKey"$\}$

- $\mathcal{L}(s_9) = \{$Response(1),"True"$\}$

- $\mathcal{L}(s_{10}) = \{$Caller_org.apache.felix.framework(6),
  Service_fr.inria.amazones.lise.app.SigApp,
  Provider_LISE-SAP-SigApp,
  Interface_fr.inria.amazones.logos.lisendium.SigAppIfc,
  Function_submitDocToUser,
  "To buy this pen at 10$ enter your PID below."$\}$

- $\mathcal{L}(s_{11}) = \{$Caller_LISE-SAP-SigApp(7),
  Service_fr.inria.amazones.lise.disp.MobileIOImpl,
  Provider_LISE-MPP-MobileIO ,
  Interface_fr.inria.amazones.logos.lisendium.MobileIOIfc,
  Function_submitDocToUser,
  "To buy this pen at 10$ enter your PID below."$\}$

- $\mathcal{L}(s_{12}) = \{$Response(7),"azer"$\}$

- $\mathcal{L}(s_{13}) = \{$Caller_LISE-SAP-SigApp(8),
  Service_fr.inria.amazones.lise.disp.MobileIOImpl,
  Provider_LISE-MPP-MobileIO ,
  Interface_fr.inria.amazones.logos.lisendium.MobileIOIfc,
  Function_submitDocToUser,
  "To buy this pen at 10$ enter your PID below."$\}$

- $\mathcal{L}(s_{14}) = \{$Response(8),"zerty"$\}$

- $\mathcal{L}(s_{15}) = \{$Caller_LISE-SAP-SigApp(9),

```
Service_fr.inria.amazones.lise.disp.MobileIOImpl,
Provider_LISE-MPP-MobileIO ,
Interface_fr.inria.amazones.logos.lisendium.MobileIOIfc,
Function_submitDocToUser,
"To buy this pen at 10$ enter your PID below."}
```

- $\mathcal{L}(s_{16}) = \{$`Response(9),"erty"`$\}$

- $\mathcal{L}(s_{17}) = \{$`Response(6),"False"`$\}$

5.3 Relation de transition

La relation de transition $\mathcal{R}$ relie les états les uns aux autres et reflète le déroulement des actions. Notre exemple ne reflète pas tout ce que l'on pourrait représenter à l'aide d'une structure de Kripke car dans cet exemple, les logs ne sont pas entrelacés et l'ordre d'exécution correspond à l'ordre d'inscription dans les logs. Concrêtement, cela signifie que nous avons $s_i \mathcal{R} s_{i+1}$ pour toute valeur de i dans $\{0, \ldots, 17\}$.

6 Modélisation de propriétés de responsabilités

Nous modélisons dans cette partie un ensemble des formules Φ de CTL qui spécifient des propriétés que l'on souhaite (ou non) voir vérifier sur la structure représentant les logs. La fonction de responsabilité en elle-même serait alors une fonction qui attribue un responsable suivant l'évaluation de Φ sur la structure. Par exemple, la fonction de responsabilité peut être définie de telle manière qu'en cas de faute correspondant à l'invalidation de $\mathcal{K} \models \Phi$, le composant c soit désigné comme responsable.

Une demande de signature doit être transmise à la carte. Considérons la propriété qui stipule que toute demande de signature doit un jour être transmise à la carte si cette demande a été transmise à l'utilisateur et qu'il a donné son accord. Sous l'angle d'une logique temporelle, cela se traduit *informellement* par :

Pour tout les chemins dans l'arbre des exécutions, si un chemin contient un état s_1 vérifiant $\ll$ la carte a présenté un document à signer à l'utilisateur $\gg$ et que par la suite il existe une exécution dans laquelle on trouve un état s_2 vérifiant $\ll$ l'utilisateur a répondu OK $\gg$, alors pour tous les chemins partant de s_2 il existe dans le futur un état s_3 vérifiant $\ll$ la

demande de signature est transmise à la carte ≫.

Pour traduire cette propriété sous la forme d'une formule CTL, nous commençons par traduire les phrases ≪ la carte a présenté un document à signer à l'utilisateur ≫, ≪ l'utilisateur a répondu ≫, ... qui sont des propriétés sur les états obtenues ici par des conjonctions de propositions atomiques prises dans la table 5.2.

≪ **la demande de signature est transmise à l'utilisateur** ≫ est vraie sur un état si la conjonction des propositions suivantes est vraie :
`Caller_LISE-SAP-SigApp(i)`,
`Service_fr.inria.amazones.lise.disp.MobileIOImpl`
`Function_submitDocToUser`,
`"To buy this pen at 10$ enter your PID below."`
(paramètres)
La conjonction de ces propositions est notée P_1.

≪ **l'utilisateur a répondu OK** ≫ est vraie s'il existe un log réponse correspondant à l'appel précédent avec la réponse ok, c'est-à-dire s'il existe un état dans lequel la conjonction des propositions atomiques suivantes est vraie :
`Response(i)`,
`"OK"` (paramètres) [1]
La conjonction de ces propositions est notée P_2.

≪ **la demande de signature est transmise à l'application de signature** ≫ est vraie sur un état si la conjonction des propositions suivantes est vraie :
`Caller_LISE-SAP-SigApp(i)`,
`Service_fr.inria.amazones.lise.card.CardImpl`
`sign`,
`"To buy this pen at 10$ enter your PID below."`
(paramètres)
La conjonction de ces propositions est notée P_3.

Finalement la propriété toute entière se traduit pas la formule suivante :

$$\Phi = P_1 \rightarrow \mathrm{A}\Box(P_2 \rightarrow \mathrm{A}\Diamond P_3)$$

La fonction de responsabilité pourrait être :

```
IF  K ⊨ Φ
THEN "pas de faute"
ELSE "l'application de signature est en faute"
```

Une demande de signature faite par le composant `org.apache.felix.framework` à la carte est toujours faite

1. ne correspond pas aux logs actuels

en utilisant un service fournit par `LISE-SAP-SigApp` Cette propriété se traduit simplement par

Φ =A□(`Caller_org.apache.felix.framework`
$\wedge$ `Service_fr.inria.app.SigApp` $\rightarrow$
`Provider_LISE-SAP-SigApp`)

Plus généralement, une plainte exprimant qu'un événement attendu ne s'est pas produit (par exemple un service n'a pas été rendu), s'exprime par une formule ϕ qui doit être vérifiée sur tous les chemins. La formule à vérifier sur les logs est donc de la forme $A\Phi$. En effet, si l'expertise montre que sur toutes les exécutions qui ont pu avoir lieu Φ est vraie alors le partie se plaignant que Φ peut légitimement obtenir gain de cause.

Au contraire, une plainte exprimant qu'un événement non autorisé a eu lieu s'exprimera par une formule de la forme $E\Phi$. Il suffit en effet que si l'expertise montre qu'il existe une exécution telle que Φ pour que la partie se plaignant de Φ soit entendue.

7 Réalisation

Nous avons réalisé un prototype d'analyseur de logs implémentant d'une part la représentation des logs sous la forme d'une structure de Kripke et d'autre part l'interrogation de ces structures par des formules écrites en CTL. Le prototype est réalisé en OCaml[7], et l'interface a été réalisée en utilisant lablgtk2. Christopher Humphries a réalisé l'interface de cet analyseur durant un stage d'été de 2 mois (stage de M1 info). Le logiciel permet de spécifier des formules en logiques temporelles via une interface graphique 6. Ces formules sont ensuite vérifiées sur un ensemble de log que peut définir l'utilisateur. Une fenêtre graphique permet de visualiser les logs en cours d'analyse, ces logs sont affichés en format *bruts* c'est à dire tels qu'ils sont effectivement dans les fichiers de logs. Une fenêtre secondaire permet aussi de visualiser une traduction de ce format *brut*.

7.1 Type des formules CTL et relation de satisfaction

Nous avons créé deux types, le premier représente les variables propositionnelles comme des chaînes de caractères :

```
type var_prop =  string
```

le second type *(formula)* représente une formule en CTL :

```
type formula =
    True
```

```
  | False
  | Neg of formula
  | VP of var_prop
  | Or of formula * formula
  | And of formula * formula
  | Imply of formula * formula
  | Equiv of formula * formula
  | A of formula
  | E of formula
  | Next of formula
  | Diamond of formula
  | Square of formula
```

Attention, toute formule CTL est un élément de type `formula` mais le contraire n'est pas vrai. Il n'y a pas de vérification au niveau du typage que la formule est bien écrite en CTL, c'est à dire des formules de la forme $QO\Phi$ où Q est un quantificateur sur les chemins qui est soit A (*pour tous les chemins*) soit E (*il existe un chemin*). O est un opérateur modal qui vaut X, $\Box$, $\Diamond$ ... Pour vérifier qu'un élément de type formule est bien en CTL, il faudra utiliser une fonction supplémentaire, ce qui n'est pas encore le cas du prototype dans sa version actuelle.

Nous vérifions si une formule est satisfaite par une structure de Kripke grâce à la fonction *satisfies* (kp :Kripke.kripke) (s :Kripke.state) (phi :formula) qui attend comme argument une structure de Kripke, un état et une formule et qui est construite par induction sur la forme de la formule phi. Remarquons que pour que les parties puissent s'accorder en toute confiance sur l'analyseur de log, cette partie du code devrait être certifiée. Nous projetons d'utiliser l'assistant à la preuve Coq pour extraire cette fonction en OCaml et vérifier que cette fonction retourne vrai si et seulement si la structure passée en argument satisfait la formule passée en argument.

```
let rec satisfies (kp:Kripke.kripke) (s:Kripke.state) (phi:formula) =
  match
    phi
  with
      True -> true
    | False -> false
    | VP(x) ->  satisfies_vp kp s x
    | Neg (psi )-> not (satisfies kp s psi)
    | Or(psi1, psi2) -> (satisfies kp s psi1) || (satisfies kp s psi2)
    | And(psi1, psi2) -> (satisfies kp s psi1) && (satisfies kp s psi2)
    | Imply(psi1, psi2) -> (not (satisfies kp s psi1))
                             || (satisfies kp s psi2)
    | Equiv (psi1, psi2) -> (satisfies kp s (Imply (psi1, psi2)))
                             && (satisfies kp s (Imply( psi2, psi1)))
```

```
    | A(psi) ->

  try
List.for_all (fun x -> satisfies kp x psi) (Kripke.nextstate s kp )
  with
      Not_found -> failwith "def de A dans la relation de satisfaction"

    | E(psi) ->
  try
    List.exists  (fun x -> satisfies kp x psi) (Kripke.nextstate s kp )
  with Not_found ->  failwith "pb sur def de E "

    | Next(psi) ->
  try
    List.for_all  (fun x -> satisfies kp x psi) (Kripke.nextstate s kp
  with Not_found ->  failwith "pb sur def de NEXT: cas des traces finie


    |Square (psi ) ->
    (List.for_all
       (fun x-> satisfies kp x (Square psi))
       (Kripke.nextstate s kp)



    |Diamond (psi ) ->
       (satisfies kp s (Next(psi)))
       || ((List.exists
     (fun x-> satisfies kp x (Square psi) )
     (Kripke.nextstate s kp)))
```

7.2 Interface graphique

La figure 6 présente l'interface graphique de l'analyseur de log réalisée par Christopher Humphries durant son stage de M1 dans l'équipe SSIR de Supélec.

Dans cette interface graphique, nous nous sommes attachés a simplifier l'utilisation du logiciel par un non-expert. Pour cela, nous avons choisi d'afficher les logs sous leur forme canonique et sous leur forme traduite en langage ≪ naturel ≫. Nous avons aussi prévu de traduire les formules CTL en langage naturel (non implémenté dans la version actuelle). L'utilisateur

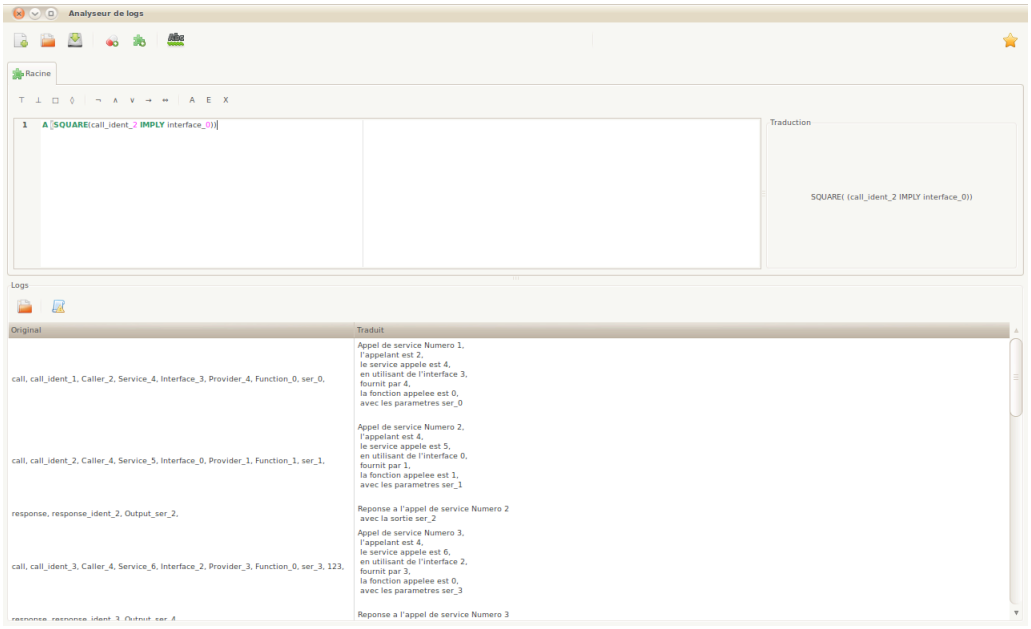


FIGURE 6 – Interface graphique de l'analyseur de log

peut construire la formule qu'il souhaite vérifier incrémentalement, chaque résultat intermédiaire pouvant ensuite être sauvegardé.

Références

[1] Rajeev Alur and Thomas A. Henzinger. Real-time logics : complexity and expressiveness. *INFORMATION AND COMPUTATION*, 104 :390–401, 1993.

[2] Batsayan Das, Dipankar Sarkar, and Santanu Chattopadhyay. Model checking on state transition diagram. *Asia and South Pacific Design Automation Conference*, 0 :412–417, 2004.

[3] Cindy Eisner, Dana Fisman, John Havlicek, Yoad Lustig, Anthony McIsaac, and David Van Campenhout. Reasoning with temporal logic on truncated paths. In *CAV*, pages 27–39, 2003.

[4] E. Allen Emerson. Temporal and modal logic. In *HANDBOOK OF THEORETICAL COMPUTER SCIENCE*, pages 995–1072. Elsevier, 1995.

[5] Stéphane Frénot. Techniques for efficient log file recording on osgi platforms. Technical report, delivrable D3.1 of the LISE project, february 2010.

[6] Daniel Le Métayer, Manuel Maarek, Valérie Viet Triem Tong, Eduardo Mazza, Marie-Laure Potet, Nicolas Craipeau, Stéphane Frénot, and Ronan Hardouin. Liability in software engineering : overview of the lise approach and illustration on a case study. In Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel, editors, *ICSE*, pages 135–144. ACM, 2010.

[7] Objective Caml. http ://caml.inria.fr/ocaml/.

[8] M. Roger, Muriel Roger, and J. Goubault-Larrecq. Log auditing through model-checking. In *In Proceedings from the 14th IEEE Computer Security Foundations Workshop (CSFW'01)*, pages 220–236. IEEE Computer Society Press, 2001.