Approche Multi-agent pour l'analyse de journaux

F. Mouysset[a,b] F. Migeon[a] M.P. Gleizes[a] C. Bortolaso[b] M. Derras[b]

[a]Institut de Recherche en Informatique de Toulouse,
Université Toulouse 3, France
{firstname}.{surname}@irit.fr

[b]Berger-Levrault
Labège, France
{firstname}.{surname}@berger-levrault.com

Résumé

À mesure que les applications se complexifient, l'usage qui en découle dévie de leur conception. Il est alors intéressant de redécouvrir des modèles de ces processus métier a posteriori, notamment en analysant les journaux d'activité des utilisateurs. Cependant, ces journaux d'activité peuvent contenir des erreurs qui compliquent la découverte de modèles fiables et réalistes. Dans cet article, un système multi-agent (SMA) appelé SAMOTRACE est conçu et s'adresse à cette problématique. Sa mise en œuvre est basée sur des agents auto-organisés. Les expériences montrent que le système tend à converger vers une solution optimale, quels que soient le type et la quantité d'erreurs présentes dans les observations.

Mots-clés : *Système Auto-Organisé, Journaux d'Evènements, Système Multi-Agent*

1 Introduction

Comme de nombreux logiciels modernes, la complexité des logiciels destinés aux services publics augmente avec l'évolution constante des réglementations et des besoins utilisateurs. Ceci a des impacts majeurs sur la qualité des logiciels et les processus de maintenance. Pour l'éditeur, cela se caractérise par un nombre croissant de bogues [11] difficilement reproductibles. De plus, l'équipe projet conçoit les tests a priori, qui peuvent ne pas refléter les usages des utilisateurs. Ainsi, un écart apparaît entre l'utilisation prévue et la réalité, conduisant à des cas inattendus, non testés et donc possiblement erronés.

Enfin, chaque évolution logicielle tend à rendre plus difficiles son utilisation et sa compréhension [12]. L'expérience utilisateur s'en trouve détériorée et le risque d'erreur augmente en conséquence.

À la lumière de ces constats, il parait clair que la qualité du logiciel doit être améliorée. L'usage effectif doit guider les travaux d'amélioration. La compréhension des usages réels devient primordiale.

La plupart des logiciels modernes consignent les actions qui sont réalisées dans des journaux. Un journal contient la séquence des actions réalisées pour chaque utilisateur. Chaque action est perçue comme un évènement. A minima, un évènement est composé du nom de la tâche effectuée, d'un horodatage et d'un identifiant utilisateur. Cependant, un même utilisateur peut exécuter plusieurs instances de cas d'utilisation en concurrence. C'est par exemple le cas lorsqu'un utilisateur ouvre plusieurs instances d'un même logiciel. Un évènement doit alors indiquer dans quelle instance de cas d'utilisation se trouve l'utilisateur. Ces informations sont appelées ID d'instance de cas d'utilisation (ou ID d'instance de processus). Un journal contenant de telles données est « étiqueté ». Il n'est pas toujours facile de déterminer à quelle instance de processus est lié un évènement. Par conséquent, dans les logiciels complexes, il est assez courant de trouver des journaux composés d'évènements sans identifiant d'instance de cas d'utilisation. Ces journaux sont qualifiés de « non étiquetés ».

En plus d'être souvent non étiquetés, les journaux d'activité utilisateur peuvent contenir des erreurs comme des évènements

manquants ou désordonnés. Enfin, le fait de ne pas étiqueter les journaux induit un « entrelacement » entre les évènements appartenant à des instances de cas d'utilisations différents. L'analyse des journaux s'en trouve compliquée.

L'état de l'art montre que les approches actuelles, de découverte de modèle de processus à partir de journaux, sont inadaptées lorsqu'elles concernent des journaux d'évènements complexes, bruités et non étiquetés. Nous avons fait l'hypothèse qu'il était alors possible de concevoir un Système Multi-Agent (SMA) capable de corriger les erreurs dans les journaux tout en découvrant de tels modèles. Cependant, cet objectif est trop difficile pour être réalisé en une seule fois. Notre conception est alors décomposée en itérations et cet article décrit la première dans laquelle le modèle de processus est supposé connu. Ainsi, la section 3 présente un SMA, appelé SAMOTRACE, conçu pour détecter et corriger (de manière optimale) les journaux. La section 4 décrit les expériences menées et analyse les résultats. Enfin, certaines limites sont pointées et nos futures itérations sont présentées.

2 Contexte et travaux connexes

Un précédent article [10] établit une revue des domaines scientifiques s'intéressant à la découverte de modèle de processus. En particulier y est présenté le process mining [1]. Celui-ci peut être divisé en deux catégories d'approches : celles qui traitent des journaux non étiquetés et celles qui traitent des journaux étiquetés. Nous avons montré que la première catégorie n'est pas souhaitable pour les journaux logiciels, en raison du peu d'informations sur le comportement réel des utilisateurs. La seconde catégorie a été expérimentée avec une approche similaire à Astromskis [3] et a montré que la complexité des journaux ne peut pas être traitée par des approches de process mining.

La communauté scientifique s'entend sur le fait que les SMA contribuent à l'analyse des données et à la découverte des connaissances [4]. Il existe plusieurs approches ayant des objectifs différents.

Abdelkafi [2] propose un modèle SMA pour la découverte de l'organisation à partir des journaux de workflow [7]. Cette approche se limite aux workflows répartis entre plusieurs acteurs, ce qui n'est pas le cas pour beaucoup de logiciels.

Les SMA peuvent également être utilisés comme outil de simulation pour améliorer les modèles de processus existants [5]. Casalicchio propose une approche basée sur les agents pour modéliser et simuler des workflows dans l'administration publique. Par cette approche, les exigences non fonctionnelles et l'équilibre entre le dimensionnement des ressources et les modifications dans le workflow sont mieux contrôlés.

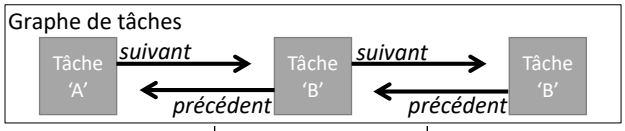
Récemment, Halaška [6] a conçu MAREA, un outil de modélisation et de simulation basé sur un SMA. L'outil est une source de données pour des activités l'analyse de process mining. De manière analogue, à Casalicchio, MAREA réalise des simulations réalistes des processus afin de tester des changements avant leur mise en œuvre. L'utilisation d'un simulateur orienté agent autorise un large éventail de comportements, et les simulations s'en trouvent plus réalistes. En particulier, les simulations ne se limitent pas aux comportements linéaires et stationnaires, mais peuvent recréer des comportements complexes et émergents.

Enfin, Clair et all [8] proposent AMAS4Opt un SMA pour l'optimisation de la fabrication des produits, plus précisément sur la planification du contrôle de fabrication et la conception de produits complexes. Cette méthode permet de résoudre des problèmes d'optimisation complexes dans un environnement dynamique. En ajoutant des comportements coopératifs entre agents, le système converge vers l'optimum.

Clair et Halaška montrent que les approches SMA sont recommandées pour les problèmes dynamiques et complexes. En outre, Halaška indique que « les processus d'entreprise peuvent également être considérés comme des systèmes très complexes ». Ainsi, les journaux produits par de tels logiciels reflètent cette complexité. Par conséquent, les SMA semblent pertinents pour le traitement des journaux logiciel, en particulier pour la découverte de modèles de processus.

2.1 Définition du problème

La résolution des erreurs contenues dans les journaux nécessite une réorganisation de l'ordre des évènements. La Fig. 1 décrit un modèle de processus et différentes traces produites par ce même modèle. Pour cet exemple, le modèle est réduit à 3 tâches en séquence, mais des modèles plus complexes sont admis (avec plusieurs tâches suivantes et précédentes, des boucles de différentes tailles…). Les journaux illustrent différentes situations possibles : journal n° 1 sans erreur, journal n° 2 un désordre local, journal n° 3 un évènement manquant. L'ordre dans lequel les évènements sont enregistrés est appelé « ordre chronologique ». Cet ordre peut être erroné, c'est pourquoi l'ordre « logique » est défini. L'ordre logique décrit des relations entre évènements qui sont conforment au graphe de tâches. Pour cela, des réorganisations peuvent être nécessaires par rapport à l'ordre chronologique, ces réorganisations définissent l'ordre logique. Par exemple, le journal n° 2 peut être corrigé en réorganisant l'ordre des évènements : l'évènement n° 4 devrait être avant l'évènement n° 3. Le journal n° 3, lui, nécessite une correction plus profonde en créant un évènement « de complétion » qui comble le manque d'évènement enregistré.
Cependant, plusieurs corrections peuvent être possibles pour un même problème. Dans le journal n° 2, deux réorganisations sont possibles, l'évènement d'index chronologique



Journal n° 1 'sans erreur'

Indice chrono-logique	Indice logique	Tâche réalisée
1	1	A
2	2	A
3	3	B
4	4	B
5	5	C

Journal n° 2 'désordonné'

Indice chrono-logique	Indice logique	Tâche réalisée
1	1	A
2	2	A
3	4	C
4	3	B
5	5	B

Journal n° 3 'évènement manquant'

Indice chrono-logique	Indice logique	Tâche réalisée
1	1	A
2	2	A
3	3	B
4	4	C
5	5	C

FIG. 1. EXEMPLE DE GRAPHE DE TACHES ET DE JOURNAUX (AVEC ET SANS ERREURS)

n° 4 est inséré soit avant l'évènement d'index chronologique n° 3, soit avant l'évènement d'index chronologique n° 5. Si l'on considère l'erreur avec le plus petit désordre (en termes de distance entre évènement) comme le plus probable alors la première réorganisation serait la plus appropriée (rasoir d'Occam). Celle-ci propose alors une solution qui minimise, au mieux, la distance entre les évènements. Si chaque évènement minimise cette distance alors elle l'est également globalement. Dans le reste de l'article, nous appelons cette métrique « distance chronologique ».

Dans cet article, les propriétés d'auto-organisation des systèmes multi-agents sont utilisées pour déterminer l'ordre logique des évènements qui optimisent cette mesure.

3 SAMOTRACE

Dans cette première version, l'objectif de SAMOTRACE est de détecter et corriger les erreurs contenues dans les journaux d'évènements. Le modèle de processus est supposé connu et correct. Ce SMA est basé sur la théorie des systèmes multi-agents adaptatifs [9]. Les propriétés d'auto-organisation sont utilisées pour explorer et converger vers l'organisation corrigeant au mieux les journaux.

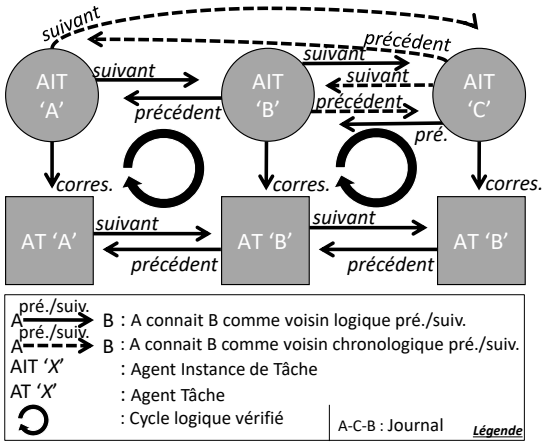


FIG. 2 – VOISINAGES ET CYCLES LOGIQUES

Les principaux concepts de la Fig. 1, sont agentifiés et représentés dans la Fig. 2. Ainsi dans SAMOTRACE on définit trois principaux types d'agents :

1) L'Agent de Tâches (AT) qui représente une tâche dans le graphe de tâches.

2) L'Agent Instance de Tâche (AIT) qui représente un évènement.

La figure 2 montre trois AT et trois évènements qui représentent, respectivement, la réalisation des tâches 'A', 'B' et 'C'. L'on dit que les AIT et AT correspondent entre eux.

3) L'Agent de Complétion (AC) qui est créé lorsqu'un évènement (ou une série d'évènements) est manquant.

Les agents sont dans un environnement social, toutes les accointances sont donc soit innées soit acquises par message. Les agents n'ont absolument pas « conscience » de l'existence du journal. Initialement, tous les AT sont créés à l'image du graphe de tâche et aucun AIT n'existe.

3.1 Agent Tâche (AT)

Une arête, dans le graphe de tâche, est une relation de voisinage donc une connaissance. Chaque AT connait ses voisins AT suivants et précédents. Les AT et leurs voisinages modélisent alors le modèle de processus. Pour l'instant, ces connaissances sont immuables et innées pour les AT, ainsi leurs relations de voisinage n'évoluent pas. La perception des AT est limitée à leurs voisins.

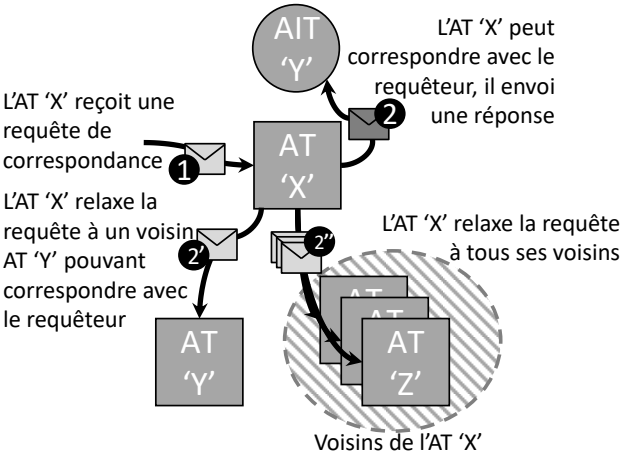


FIG. 3 – COMPORTEMENT DE L'AT

Le rôle de l'AT est d'aider les AIT à trouver l'AT avec lequel correspondre. Pour cela, les AIT envoient des requêtes spécifiques. La Fig. 3 détaille le comportement d'un AT. En ❶, l'AT 'A' reçoit une requête. Cette requête peut provenir d'un autre AT ou d'un AIT (voir 3.2). Trois possibilités :

- ❷ l'AT peut correspondre à l'AIT demandeur, il lui répond alors.
- ❷' l'AT connait un voisin AT qui peut correspondre avec le demandeur.
- ❷'' l'AT ne connait pas d'agents capables de satisfaire la requête, il la relaxe alors à tous ses voisins.

Le mécanisme de relaxation est limité pour éviter les problèmes d'inondation et d'auto-entretien de messages.

3.2 Agent Instance de Tâche (AIT)

L'AIT possède un voisinage dit « chronologique » qui reflète l'ordre chronologique des évènements dans le journal (Fig. 2 flèches pointillés). Ce voisinage est inné et immuable. À la lecture d'un évènement du journal, un AIT est créé et son voisinage chronologique est initialisé conformément à l'ordre du journal ; et il est mis en relation avec un AT candidat. L'objectif principal des AIT est de trouver et de corriger au mieux l'ordre chronologique c'est pourquoi on définit également le voisinage logique. Dans la Fig. 2 le voisinage logique décrit un ordre alternatif correct. Ce voisinage est initialisé avec le voisinage chronologique.

Pour atteindre son but l'AIT cherche à minimiser sa criticité qui représente la distance à son but. La criticité d'un AIT est un leximin de 3 critères : 1) connaissance d'un AT avec lequel correspondre, 2) connaissance d'AIT vérifiant les cycles logiques et 3) connaissances d'AIT optimisant la distance chronologique. Pour chaque critère, l'AIT dispose de stratégies basées sur la réorganisation de son voisinage logique. Le premier critère doit être vérifié pour passer aux suivants.

1) Trouver sa correspondance avec l'AT

À la mise en relation avec un AT, l'AIT vérifie sa correspondance. Soit l'AIT a trouvé son AT et met à jour sa connaissance de correspondance. Soit, l'AIT envoie une requête de correspondance à l'AT, pour diffusion aux autres AT. Cette partie du comportement est spécifiée sous forme de règles appelées « règles de correspondance ».

L'AIT est un agent coopératif, donc toute information pouvant aider ses voisins doit être partagée avec eux. C'est le cas lorsqu'un AIT trouve l'AT avec lequel correspondre. En effet, les voisins de cet AT peuvent correspondre avec les voisins de l'AIT. Par exemple, dans la Fig. 4 (gauche), en ❶, l'AIT 'A' reçoit une réponse de correspondante par l'AT 'A'. Il met donc à jour sa connaissance de correspondance ❷①. L'AIT 'A' peut alors aider son voisin l'AIT 'B'. Pour cela, l'AIT 'A' perçoit les voisins suivants de l'AT 'A', il dispose alors de toutes les informations pour déterminer s'il existe une correspondance entre l'AIT 'B' et un des voisins. Si l'AT 'B' correspond, l'AIT 'A' informe alors l'AIT 'B' en ❷②. Ainsi, l'AIT 'A' a aidé l'AIT 'B' à trouver sa correspondance. L'AIT 'B' met à jour ses connaissances ❸, il peut alors aider ses voisins à son tour.

2) Détection et correction d'erreurs dans le journal

Le deuxième objectif d'un AIT est de détecter et corriger les erreurs du journal. Pour cela, l'AIT vérifie ses deux cycles logiques. Un AIT vérifie un cycle logique si l'AT avec lequel il correspond a un AT voisin qui est l'AT de son AIT voisin. La Fig. 2 illustre le concept de cycles logiques. Notons qu'un AIT doit vérifier (au pire) deux cycles logiques, un précédent et un suivant.

Dans la Fig. 4 (gauche) le cycle logique est vérifié par l'AIT 'A' après sa correspondance avec AT 'A'. En revanche, dans la Fig. 4 (droite), le cycle logique est invalidé par l'AIT 'A'. Il invalide sa connaissance sur son voisinage logique suivant ❷③ et transmet cette invalidation à l'AIT 'C' ❷② (règle d'invalidation). À la réception de cette information, l'AIT 'C' invalide sa connaissance sur son voisinage logique précédent avec l'AIT 'A' ❸. Finalement, l'AIT 'A' et l'AIT 'B' ont chacun un voisin logique manquant. La stratégie pour retrouver un nouveau voisin logique dépend du type d'erreur.

Le premier type d'erreur est le désordre local. Ainsi, un AIT recherche localement un autre AIT pouvant vérifier le cycle logique avec lui. Pour cela, l'AIT envoie à ses voisins chronologiques une requête « de recherche d'un voisin logique ». Cette requête est traitée par les voisins qui déterminent s'ils peuvent devenir ce nouveau voisin. Si c'est le cas, l'AIT informe le demandeur et met à jour ses connaissances sur son voisinage logique. Sinon la requête est relaxée auprès de l'autre voisin qui évaluera la demande à son tour... Comme l'erreur est supposée être locale, la requête est limitée en nombre de relaxations. Le comportement décrit ici forme la « règle de recherche de voisin logique local ». Lorsqu'un requêteur envoie une requête, il définit le nombre de relaxations en fonction du délai de réponse estimé. Lorsque le délai

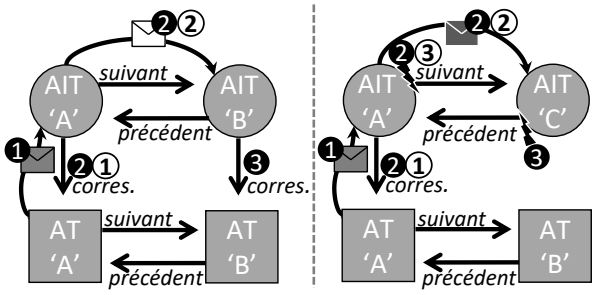


FIG. 4 – VALIDATION (GAUCHE) OU INVALIDATION (DROITE) DE CYCLE LOGIQUE

de réponse expire, l'agent considère le second type d'erreur.

Le deuxième type d'erreur est le désordre distant. Cela peut être dû à l'entrelacement d'évènements appartenant à des instances de cas d'utilisation différents. Par conséquent, certains AIT ont des voisins logiques manquants et ces AIT devraient être voisins logiques mutuellement. L'entité de liaison (EL) est une entité facilitatrice qui met en relation ces agents. Ainsi, un AIT contacte l'EL lorsqu'il ne parvient pas à résoudre une erreur locale. Lorsque l'EL reçoit cette demande, elle la mémorise. Également, elle retourne au demandeur une liste de toutes les requêtes compatibles précédemment mémorisées. Le demandeur reçoit alors tous les identifiants des agents susceptibles de vérifier le cycle logique avec lui, le meilleur est choisi (détaillé dans le paragraphe *3*). Ce comportement est régi par les « règles protocolaires avec l'entité de liaison ».

Le troisième type d'erreur est envisagé lorsqu'aucun nouveau voisin n'est trouvé. Dans ce cas, il ne s'agit plus de réorganiser les voisinages logiques, mais plutôt de créer un voisin logique manquant. Ainsi, l'AIT crée un agent de complétion (AC). Le rôle de l'AC est de compléter le journal lorsqu'il contient des évènements manquants. Dans la Fig. 5, l'évènement matérialisant la réalisation de la tâche 'B' est absent. Pour vérifier leurs cycles logiques les AIT 'A' et 'B' doivent créer un

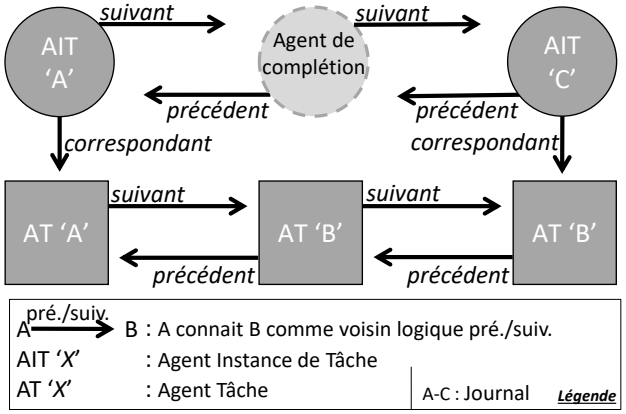


FIG. 5 – L'AGENT DE COMPLETION RESTAURE LA CONNEXITE DES EVENEMENTS

AC. Cependant, un AIT préfère toujours vérifier un cycle logique avec un autre AIT qu'avec un AC. Ainsi, si par la suite, l'AIT 'A' et/ou AIT 'C' trouvent (via l'entité de liaison) un AIT 'B', ils pourront se défaire de l'AC pour réorganiser leurs voisinages avec l'AIT 'B'. La règle de « création d'AC » est la dernière partie du comportement des AIT.

3) Optimisation de l'ordre logique

Le troisième et dernier objectif vise à éviter les minimums locaux en minimisant les distances chronologiques entre deux AIT. En effet, plusieurs AIT peuvent satisfaire le même cycle logique. Par exemple, Fig. 6, l'AIT 'B' reçoit une demande de recherche de voisin manquant ❶. Il a le choix entre deux AIT 'A', l'un avec un index chronologique de 1 et l'autre de 3. Pour déterminer quel AIT minimise la distance chronologique, l'AIT 'B' anticipe les conséquences du remplacement de son voisin logique courant par le candidat ❷. Il calcule les distances chronologiques pour chacun des agents candidats. La distance chronologique avec le voisin courant et AIT 'B' est de 4, tandis que la distance entre le candidat est de 2. La distance chronologique anticipée diminue par rapport à la distance actuelle. Le remplacement du voisin courant par le candidat est coopératif. Pour cela, AIT 'B' invalide sa connaissance de voisinage logique avec l'AIT 'A' 1 ❸① et lui envoie un message d'invalidation ❸②. Dans le même temps, il envoie un message de validation à AIT 'A' 3 ❸③. Ici, l'anticipation permet

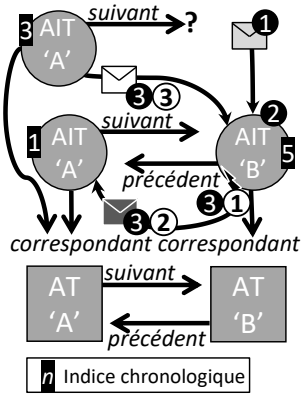


FIG. 6 – OPTIMISATION DU VOISINAGE LOGIQUE

d'estimer les effets du remplacement sur la distance chronologique. Mais ce n'est qu'une composante de l'anticipation. Tous les critères du leximin sont utilisés.

4 Expérimentations

Nous avons réalisé deux séries d'expériences pour valider le comportement du SMA. Chacune des expériences nécessite un journal d'évènements et un graphe de tâches comme entrées. La sortie est le journal réordonné et les criticités résiduelles. L'ensemble est comparé à un test de référence.

La première série d'expériences est constituée de tests fonctionnels. Chaque test aborde une partie spécifique du problème à résoudre. Comme l'indique le Tableau I, ces tests sont ad hoc et minimaux (graphes de tâches et journaux minimaux), mais ils demeurent représentatifs des erreurs possibles.

De plus, la simplicité des tests permet une approche de conception incrémentale du système. Chaque incrément se préoccupe d'une nouvelle fonctionnalité, c'est-à-dire d'un nouveau type d'erreur. L'ajout d'une fonctionnalité se traduit par l'évolution et l'ajout de règles. L'ensemble des tests est continuellement joué afin d'y détecter, au plus tôt, une éventuelle régression. Bien qu'incrémentale, cette conception est basée sur une approche de développement pilotée par les tests. Enfin, notons que l'utilisation de tests simples facilite l'explicabilité et le débogage du système (à une échelle réduite).

La deuxième catégorie d'expériences comprend les tests non fonctionnels de passage à l'échelle, ils impliquent des graphes de tâches et des journaux plus conséquents. Les données en entrée sont générées automatiquement selon différentes configurations. Par exemple, la synthèse des graphes de tâches est paramétrée par le nombre de tâches et leur connectivité moyenne. Puis, des marches aléatoires dans ces graphes produisent les journaux d'évènements. Cette étape est paramétrée par le nombre d'évènements souhaités. Certains journaux sont modifiés manuellement pour ajouter les différents types d'erreurs.

Chaque test est exécuté plusieurs fois avec différente configuration listée dans le tableau II. Par exemple, le test n° 1 a 12

TABLEAU II : TESTS DE PASSAGE A L'ECHELLE

#	**Objectif du test**	**Caractéristiques du graphe de tâches**	**Caractéristiques du journal**	**Résultat attendu**
1	Graphes de tâches de taille moyenne et pas d'erreur dans les journaux	Nombre de tâches : 50, 100, 500, 2500 Connectivité moyenne : 2 to 5	Nombre d'évènements : entre 50-70	1 solution optimale sans criticité résiduelle
2	Graphe de tâche important sans erreur	Nombre de tâches : 10.000 Connectivité moyenne : 2	Nombre d'évènements : 500-2.500	1 solution optimale sans criticité résiduelle
3	n° 1 et n° 2 avec erreur dans les journaux	n° 1 et n° 2	Différents types d'erreurs ajoutées manuellement	Des criticités résiduelles les plus petites possibles

TABLEAU I : TESTS FONCTIONNELS

#	**Objectif du test**	**Règles impliquées**	**Caractéristiques des entrées**	**Journal réordonné attendu**	**Criticité résiduelle attendue**
1	Cas parfait	Règles de correspondance et vérification de voisinage logique	Graphe de tâches tel que défini dans la Fig. 1 et journal n° 1 (Fig. 1)	*A-B-C*	Pas de criticité résiduelle
2	Auto-organisation pour la résolution de désordre local	Règles n° 1 + règle de vérification de voisinage logique + règle de recherche de voisin logique local	Graphe de tâches tel que défini dans la Fig. 1 et journal de 3 évènements avec un désordre local (B et C) : A-C-B	A-B-C	distance chronologique =6
3	Auto-organisation pour la résolution de désordre distant	Règles n° 2 + règles protocolaires avec l'*entité de liaison*	Graphe de tâches tel que défini dans la Fig. 1 et journal de 6 évènements avec un désordre distant : A-C-A-C-B-B	A-B-C A-B-C	distance chronologique =24
4	Évolution pour la résolution d'évènement manquant	Règles n° 3 + règle de création d'*agent de complétion*	Graphe de tâches tel que défini dans la Fig. 1 et journal de 2 évènements avec un évènement manquant : A-C	A – α – C (α *agent de complétion)*	1 *agent de complétion*

[a.] Pour créer un désordre distant, exceptionnellement pour ce cas de test, le nombre de relaxations est égal à 1.

configurations (3 x 4). Chaque graphe de tâches est utilisé pour générer 50 journaux de différentes tailles (fonction de la taille du graphe de tâches).

Chaque configuration de test est alors exécutée plusieurs fois. Contrairement à un agent qui possède un nombre limité d'états, un SMA, par la combinatoire de tous les agents, décrit un ensemble très important d'états. Tester tous ces états est impossible. Pire, à cause de la nature dynamique et complexe du système, il est impossible de prouver la convergence dans le cas général. Nous ne pouvons valider le système que par un grand nombre d'expérimentations.

Les tests sont effectués sur une machine standard (Intel® Core™ i7-7500U CPU @2.70GHz - RAM 16 Go). Le SMA est implémenté en Java (JDK 1.8).

Chaque test fonctionnel est exécuté 500 fois. Une exécution dure entre 900 ms et 1,5s et nécessite entre 60 et 120 cycles d'exécution.

4.1 Résultats

Tous les tests fonctionnels et de passage à l'échelle ont été validés.

1) Résultats des tests fonctionnels

L'expérimentation débute avec toutes les règles désactivées. L'ensemble minimal de règles est ajouté pour rendre le système fonctionnel. Des erreurs sont ensuite ajoutées afin de démontrer l'insuffisance des règles activées. De nouvelles règles sont alors ajoutées et ainsi de suite jusqu'à ce que toutes les typologies d'erreurs soient résolues.

L'exécution du test n° 1 sans aucune règle n'entraine aucune résolution. Le nombre d'AIT sans correspondance demeure élevé et inchangé (égal au nombre d'évènements). La Fig. 7 montre l'évolution des criticités moyennes et leurs enveloppes de variation avec l'activation des règles de correspondance (uniquement). Les 3 AIT trouvent leur correspondant AT. De plus, pour un journal sans erreur, aucune règle supplémentaire n'est nécessaire pour la vérification des voisinages

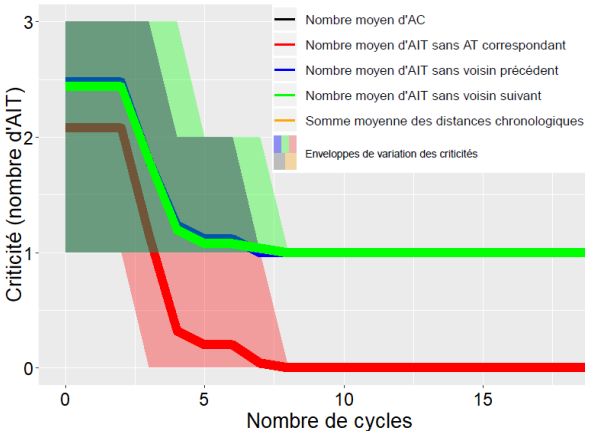


FIG. 7 – JOURNAL SANS ERREUR (TEST N°1, REGLES DE CORRESPONDANCE ACTIVEE)

logiques. Seuls, les deux voisinages logiques sont manquants aux extrémités du journal. AIT 'A' n'a pas de précédent et AIT 'C' n'a pas de suivant.

Puisque le système est fonctionnel, un désordre local est ajouté. Sur la Fig. 8 (haut), le nombre de voisins logiques manquants est de 6 (3 agents sans voisins précédents et suivants). En activant les règles associées au test n° 2, les désordres locaux sont résolus. Dans la Fig. 8 (en bas), le nombre de voisins logiques manquants est égal à 2 (toujours dû

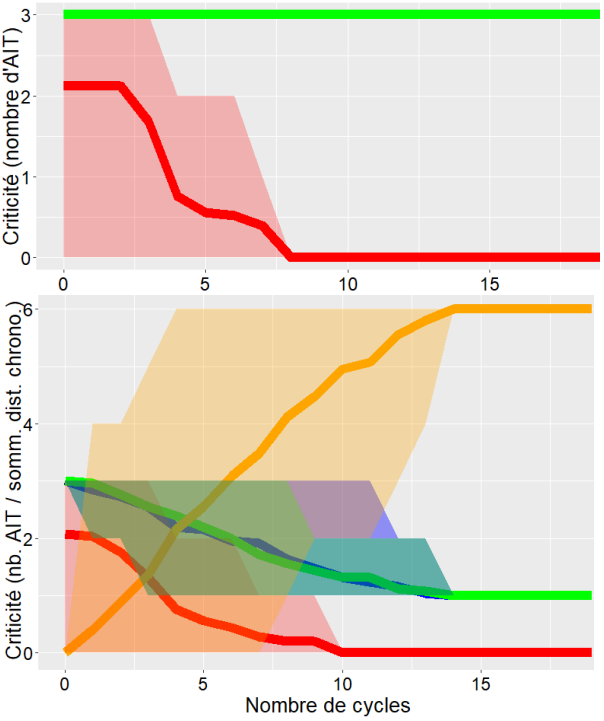


FIG. 8 – JOURNAL AVEC UN DESORDRE LOCAL (TEST N° 2). HAUT SANS LES REGLES, BAS AVEC LES REGLES

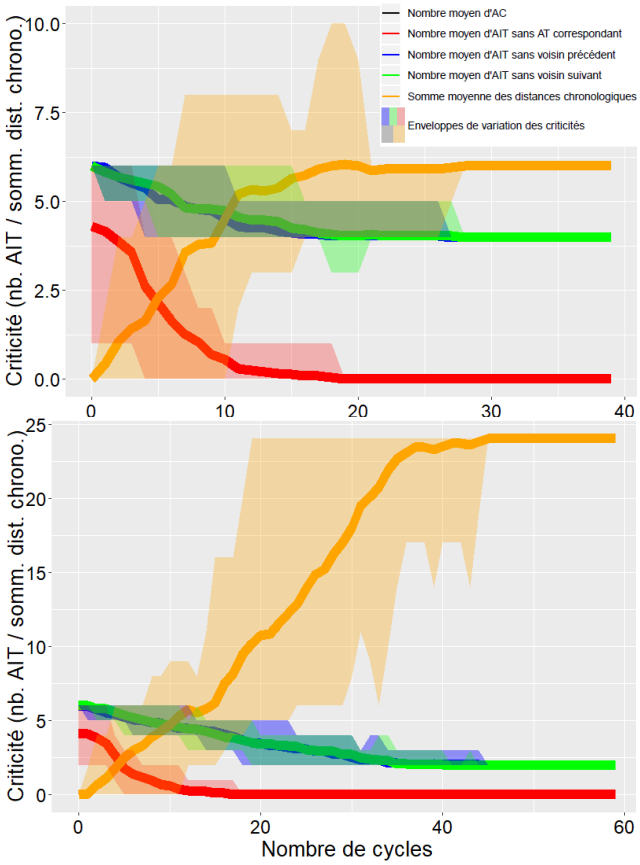


FIG. 9 – JOURNAL AVEC UN DESORDRE DISTANT (TEST N°3). HAUT SANS LES REGLES, BAS AVEC LES REGLES.

aux extrémités du journal). La somme des distances chronologiques augmente parce que les nouveaux voisinages logiques sont différents du voisinage chronologique.

Le test n° 3 débute par l'ajout d'un désordre distant, l'ensemble de règles est inchangé. Il en résulte des criticités résiduelles, Fig. 9 (haut), où 4 voisins logiques sont manquants. L'activation des règles du test n° 3, Fig. 9 (bas), résout le désordre distant. Mais, à nouveau, cela se traduit par une augmentation de la distance chronologique.

Enfin, dans le test n° 4 le journal contient un évènement manquant. Sans règle supplémentaire, aucun des voisinages logiques n'est vérifié. En revanche, l'activation de la règle de création d'AC résout ce problème. Dans la Fig. 10, nous observons que contrairement aux tests précédents, le nombre de voisins logiques manquants est de 0. En plus, le nombre d'AC n'est pas de 1, comme attendu, mais de 2. En fait, la possibilité de créer des AC a permis de relier les deux extrémités du journal via un AC. Le résultat du test est donc correct.

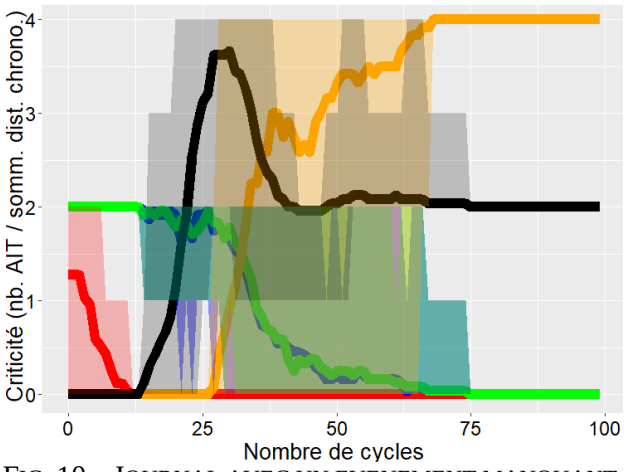


FIG. 10 – JOURNAL AVEC UN EVENEMENT MANQUANT (TEST N° 4 REGLES ACTIVEES).

2) *Résultats du passage à l'échelle*

Pour ces tests, les données en entrées sont générées et les erreurs sont ajoutées manuellement. Deux tests sont ici détaillés. Le premier est constitué d'un graphe de 10 tâches et d'une connectivité moyenne de 5. Le journal généré contient 1000 évènements et 10 erreurs de chaque type sont ajoutées. Le second test décrit une expérience avec 1000 tâches avec une connectivité moyenne de 5. Le journal est composé de 786 évènements, avec les mêmes types de bruit. Chaque test est exécuté 25 fois (avec les mêmes données en entrées). Pour ces 25 runs, l'évolution des criticités au cours de la résolution est retracée respectivement FIG. 11 et Fig. 12.

Dans les deux figures, le nombre d'AIT sans correspondant et le nombre de voisins logiques manquants diminuent ❶. Pour satisfaire les cycles logiques, des AC sont créés. Par conséquent, le nombre de voisins logiques manquants diminue à mesure que le

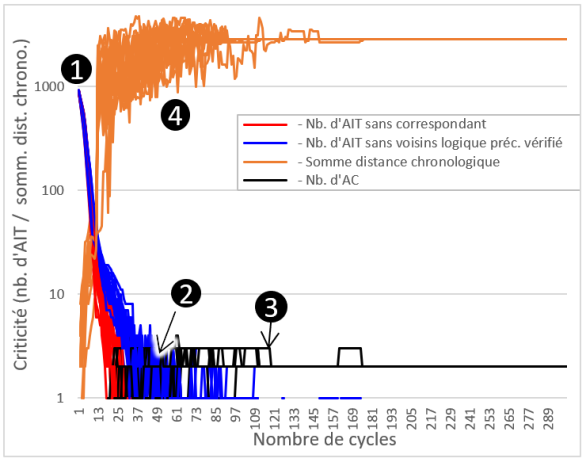


FIG. 11 – ÉVOLUTION DE LA CRITICITE – 10 TACHES ET JOURNAL AVEC ERREURS

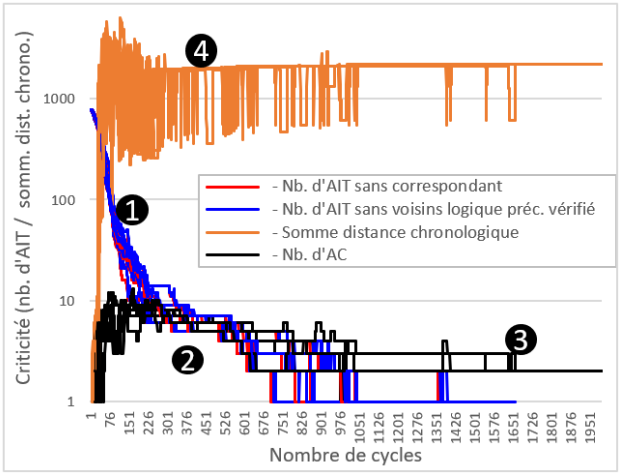


FIG. 12 – ÉVOLUTION DE LA CRITICITE – 1000 TACHES ET JOURNAL AVEC ERREURS

nombre d'AC augmente ❷. Après quelques réorganisations et contacts avec l'EL, plusieurs AIT se séparent de leur AC ❸ (diminution du nombre d'AC). Cependant, la distance chronologique augmente ❹. Enfin, le système se stabilise.

Pour toutes les exécutions, le système est parvenu à converger vers un résultat.

5 Conclusion et perspectives

Nous avons fait l'hypothèse que l'approche SMA pourrait être adaptée à l'analyse de journaux d'activité d'utilisateur sur des logiciels, en particulier pour la découverte de modèles de processus. Afin de valider cette hypothèse, nous avons conçu et mis en œuvre SAMOTRACE, un système multi-agent auto-organisé et adaptatif.

Cet article présente la première itération de notre travail : la résolution des erreurs contenues dans les journaux. Cette approche a été validée par deux types de tests. Les tests fonctionnels montrent la cohérence de notre base de règles. Tandis que les tests de passage à l'échelle montrent la viabilité de l'approche appliquée aux journaux de taille plus conséquents. Les tests ont démontré la viabilité de notre approche de correction des journaux logiciels.

Nos futurs travaux vont ajouter de nouvelles règles d'auto-organisation aux AT. Par exemple, certaines relations entre les AT seront supprimées, et les AT devront être en mesure de se réorganiser pour réparer les erreurs. Néanmoins, les règles devront être conçues de manière à maintenir la coopération entre les AIT et les AT. En effet, lorsque le TIA détecte une erreur, celle-ci peut provenir d'une erreur dans le journal ou dans le voisinage des TA.

Enfin, dans notre protocole expérimental certaines notions mériteraient d'être précisées, notamment la correspondance entre AIT et AT. Pour l'instant, une simple égalité Leibnizienne est utilisée. Pour finir, nous nous adressons le problème de la génération automatique de journaux bruités dans le prochain jalon.

References

[1] W. van der Aalst, "*Data Science in Action*", Springer Berlin Heidelberg, 2016.

[2] M. Abdelkafi, L. Bouzguenda, and F. Gargouri, "DiscopFlow: A new Tool for Discovering Organizational Structures and Interaction Protocols in WorkFlow", 2012.

[3] S. Astromskis, A. Janes, and M. Mairegger, "A process mining approach to measure how users interact with software: an industrial case study", Proceedings of the *International Conference on Software and System Process*, 2015.

[4] L. Cao, V. Gorodetsky, and P. A. Mitkas, "Agent Mining: The Synergy of Agents and Data Mining," *IEEE Intelligent Systems*, vol. 24, no. 3, pp. 64–72, 2009.

[5] E. Casalicchio and S. Tucci, "Public Administration Workflows Re-engineering : An Agent-Based & Approach", In : *Multiagent Systems and Applications*, Springer Berlin Heidelberg, 2013.

[6] M. Halaška and R. Šperka, "Advantages of application of process mining and agent-based systems in business domain", In *KES International Symposium on Agent and Multi-Agent Systems: Technologies and Applications*, pp. 177–186, 2018

[7] D. Hollingsworth, D. Hollingsworth, "The Workflow Reference Model: 10 Years On", In : *Fujitsu Services, UK; Technical Committee Chair of WfMC*, pp. 295–312, 2004.

[8] Gaël Clair, Elsy Kaddoum, Marie-Pierre Gleizes, Gauthier Picard. "Self-Regulation in Self-Organising Multi-Agent Systems for Adaptive and Intelligent Manufacturing Control", In *IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, IEEE Computer Society, 2008.

[9] G. Di Marzo Serugendo, M.-P. Gleizes, and A. Karageorgos, "*Self-organising software : from natural to artificial adaptation*". Springer, 2011.

[10] F. Mouysset et al., "Investigations of Process Mining Methods to discover Process Models on a Large Public Administration Software," *INFORSID*, 2019.

[11] R. Subramanyam and M. S. Krishnan, "Empirical analysis of CK metrics for object-oriented design complexity: implications for software defects", In *IEEE Transactions on software engineering*, vol. 29, no. 4, pp. 297–310, 2003.

[12] D. V. Thompson, R. W. Hamilton, and R. T. Rust, "Feature Fatigue: When Product Capabilities Become Too Much of a Good Thing", In *Journal of marketing research*, vol. 42, no. 4, pp. 431–442, 2005.