Imposition de propriétés temporisées à l'exécution

Yliès Falcone

Imposition de propriétés temporisées à l'exécution

Yliès Falcone

Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble, France
ylies.falcone@univ-grenoble-alpes.fr - www.ylies.fr

Résumé L'imposition de propriétés (en anglais *runtime enforcement*) désigne les méthodes, techniques et outils permettant de garantir que l'exécution d'un système est conforme à sa spécification. Cet article présente brièvement certains des résultats majeurs obtenus lors des travaux de ce domaine de recherche.

1 Introduction

Évaluer complètement la fiabilité d'un système logiciel durant son développement reste un problème difficile, voire infaisable pour plusieurs raisons bien connues. Par exemple, la taille de l'espace des comportements, des configurations possibles, et de leur combinaisons rendent le problème infaisable ; même avec les capacités de calcul modernes. De plus, les scénarios d'utilisation réels du système logiciel peuvent être inconnus ou difficilement productibles lors du développement.

Pour améliorer la fiabilité des logiciels durant leur exploitation, ceux-ci peuvent être associés à des solutions permettant d'éliminer et de tolérer les fautes logicielles. Durant l'exécution, ces solutions essayent d'une part de détecter les fautes qui n'ont pas été trouvées durant le développement (lorsque ces fautes produisent des effets observables) et de permettre au système de fonctionner correctement malgré les fautes.

Dans cet article, nous nous intéressons aux solutions dites d'*imposition de propriétés durant l'exécution* des systèmes logiciels [14,28,31,49]. La démarche employée est d'exprimer et formaliser les bons comportements du système (extraits par exemple de sa spécification) sous forme de propriétés en faisant l'hypothèse que les fautes correspondent à des violations de ces propriétés. Les propriétés sont généralement exprimées dans des formalismes à base de logique, de machine à états, d'algèbre de processus, etc. Il faut entendre la notion de propriété au sens large ; une propriété pouvant se comprendre comme un prédicat sur la séquence d'états traversés par le programme ou la séquence d'actions qu'il a produites. Considérons deux exemples très simples pour illustrer la notion de propriété :

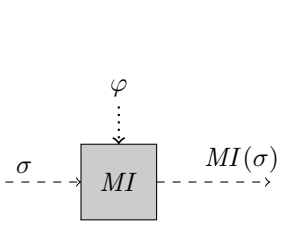
— Pour un système devant journaliser certaines opérations sensibles, une propriété peut décrire les exécutions où les entrées du journal comportent des identificateurs croissants et ne comportent pas d'informations confidentielles.
— Pour un système à base de tâches, une propriété peut décrire des exécutions où les tâches s'exécutent selon un certain ordre et de manière équitable.

De manière plus générale, l'imposition à l'exécution peut être utilisée dans une variété de domaines d'application. Par exemple, les techniques d'imposition peuvent être utilisées :

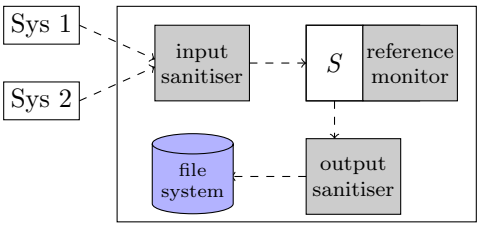
— dans les domaines de la sécurité et des réseaux en filtrant les paquets et commandes reçus par les entités du système ;
— pour prévenir les dénis de service en assurant un laps de temps minimum entre le traitement de requêtes ;
— pour assurer la bonne composition de service pour assurer les pré-conditions d'utilisation d'un service ;
— à des fins d'ordonnancement de tâches en temps-réel pour assurer des propriétés comme l'absence de blocage ou interblocage et la vivacité du système.

Structure de l'article. Le reste de cet article est structuré comme suit. La section 2 introduit l'imposition à l'exécution de manière générale. La section 3 se consacre aux nombreux travaux visant à imposer des propriétés non temporisées. La section 4 traite de travaux plus récents (et moins nombreux) visant à imposer des propriétés temporisées. La section 5 conclut.

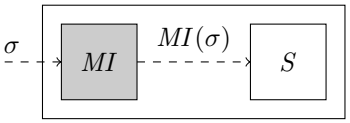
Remarque 1 *Dans la suite de cet article, nous éviterons de traduire certains termes anglais en français pour éviter les approximations de sens induites par la traduction et faciliter la recherche de documentation (principalement en anglais) sur les termes utilisés. De plus, étant donnée la variété de formalismes discuté dans cet article, nous ne donnerons pas les définitions formelles sous-jacentes aux concepts discutés et indiquerons plutôt des références.*



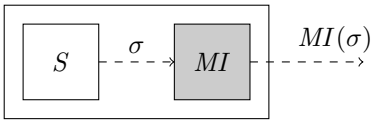
(a) Vue conceptuelle du problème d'imposition à l'exécution



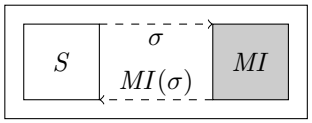
(b) Scénario illustrant les utilisations possibles d'un mécanisme d'imposition



(c) *MI* comme input sanitiser



(d) *MI* comme output sanitiser



(e) *MI* comme reference monitor

FIGURE 1: Illustration du problème de l'imposition à l'exécution : un mécanisme d'imposition *MI* transforme une entrée σ en une sortie $MI(\sigma)$ selon une propriété φ et pour un système S.

2 Préliminaires et définitions

Objectifs de l'imposition à l'exécution. En tant que domaine de recherche, l'imposition de propriétés à l'exécution [14,28,31,49] (en anglais *runtime enforcement of properties*) est une « branche » du domaine plus général de la vérification de propriétés à l'exécution [1,17,25,30] (en anglais runtime verification of properties). Le lecteur pourra consulter les articles [14,28] qui présentent une bonne partie des travaux dans le domaine de l'imposition de propriétés jusqu'à leur date de publication. Alors que la vérification à l'exécution s'intéresse généralement au problème de l'oracle sur les exécutions (c'est-à-dire à déterminer si l'exécution courante est conforme à des propriétés), le problème principal de l'imposition à l'exécution est d'anticiper et réagir aux fautes en modifiant directement l'exécution du logiciel si nécessaire.

Abstraction du système. Les travaux de recherche en imposition à l'exécution s'abstraient généralement des détails d'implémentation et plus précisément de la *manière* dont la spécification est imposée sur le système. C'est-à-dire, l'existence de capteurs et d'actionneurs est généralement supposée ; ceux-ci étant obtenus par instrumentation. De plus, les problèmes en imposition à l'exécution concernent principalement la définition d'une relation d'entrée-sortie sur les séquences d'événements ou d'actions du système (voir Figure 1a) : étant donnée une séquence d'actions σ que le système souhaite produire, comment le mécanisme d'imposition *MI* peut la transformer en une séquence $MI(\sigma)$ conforme au comportement attendu décrit par une propriété φ. C'est-à-dire, l'exécution réelle est abstraite en une séquence d'événements pris dans un alphabet Σ qui influencent la satisfaction de la propriété φ (Ainsi, la propriété φ est un ensemble de séquences sur Σ admissibles). La séquence d'événements est obtenue en instrumentant le système avec du code permettant d'observer l'occurence de chaque événement de Σ dans le système. Plus précisément, un cadre d'imposition à l'exécution doit décrire comment transformer une séquence d'événements d'entrée (possiblement incorrecte) en une séquence d'événements de sortie à l'aide d'un *mécanisme d'imposition* [1]. La transformation est dirigée par la propriété φ qui est utilisée directement pour synthétiser le mécanisme d'imposition.

Hypothèses et architectures. Avant d'élaborer sur les différentes manières par lesquelles un mécanisme peut transformer la séquence d'entrée et comment le mécanisme d'imposition peut être synthétisé (dans les sections suivantes), nous donnons certaines des hypothèses implicites faites en imposition à l'exécution. En particulier, il est important de noter que la présentation conceptuelle dans la Figure 1a abstrait plusieurs architectures possibles.

1. Nous suivons la terminologie de [19] qui généralise les terminologies précédemment utilisées. Nous utilisons le terme de mécanisme d'imposition en englobant les définitions de mécanismes dédiés à l'imposition et décrits à différents niveaux d'abstraction. De plus, le terme mécanisme d'imposition nous permet d'abstraire l'architecture effective du mécanisme et son placement par rapport au système vérifié.

Nous présentons et illustrons quelques exemples d'architectures plus concrètes sur un scénario d'utilisation dans l'Exemple 1. Tout d'abord, un mécanisme d'imposition peut être utilisé pour faire de l'*input sanitisation* (voir Figure 1c). Dans ce cas, le mécanisme est utilisé pour « protéger » le système de son environnement non sûr. Toutes les entrées du système doivent d'abord passer par le mécanisme d'imposition qui filtrent celles qui pourraient endommager le système. Par exemple, le mécanisme d'imposition peut-être utilisé comme coupe-feu ou pour assurer que les préconditions requises pour utiliser un système sont remplies lors de la composition avec d'autres systèmes. Aussi, un mécanisme d'imposition peut être utilisé à des fins d'*output sanitisation* (voir Figure 1d). Toutes les sorties du système doivent d'abord passer par le mécanisme qui les filtre ou les transforme. Par exemple, le mécanisme peut être utilisé pour prévenir la fuite d'informations ou pour transformer la trace produite par le système. Enfin, un mécanisme d'imposition peut être utilisé comme *reference monitor* (voir Figure 1e). Toutes les actions d'intérêt ou les changements d'état importants sont d'abord soumis à l'approbation du mécanisme d'imposition. Par exemple, le mécanisme peut réguler l'accès à des primitives sensibles ou des opérations système.

Exemple 1 (Utilisations d'un mécanisme d'imposition) *Considérons le système S représenté dans la Figure 1b où des mécanismes sont utilisés pour imposer un comportement correct et assurer la qualité à l'exécution. Supposons que le but de S soit de réaliser un comportement basé sur des services fournis par des systèmes externes Sys1 et Sys2. Les actions de S sont commandées par des utilisateurs (non représentés dans la Figure 1b) et l'exécution de chaque action doit être journalisée dans le système de fichier. L'input sanitiser est utilisé pour transmettre à S l'information d'entrée seulement lorsque Sys1 et Sys2 tout deux fournissent le service attendu. Le reference monitor est utilisé pour surveiller les actions importantes de S en empêchant S d'exécuter des actions lorsque celles-ci ne sont pas autorisées. L'output sanitiser est utilisé pour s'assurer que les actions sont enregistrées de manière appropriée en garantissant un format de journal prédéfini, en rendant anonyme l'information sensible de l'utilisateur ou en supprimant l'information superflue.*

Exigences sur les mécanismes d'imposition. Dans la plupart des travaux en imposition à l'exécution, la relation d'entrée-sortie réalisée par le mécanisme d'imposition doit satisfaire les exigences suivantes :

- *Correction (en anglais soundness) :* la séquence de sortie ($MI(\sigma)$) doit être correcte par rapport à la spécification.
- *Transparence (en anglais transparency) :* une séquence d'entrée correcte ($\sigma \in \varphi$) ne doit pas être modifiée, si possible. [2]

La correction et la transparence sont deux propriétés complémentaires permettant d'assurer une certaine qualité de service au mécanisme d'imposition. La correction assure que la spécification sera respectée ; d'une certaine manière elle restreint les séquences que peut produire le mécanisme d'imposition. La transparence assure une certaine "fidélité" à ou "proximité" avec la séquence d'entrée (c'est-à-dire la séquence que souhaitait produire le système) ; d'une certaine manière elle contraint le mécanisme d'imposition à produire des séquences les plus proches possibles de celle d'entrée (au lieu de simplement produire la plus petite séquence correcte qui peut être de longueur nulle). La transparence s'exprime généralement à l'aide d'une relation (d'équivalence ou d'ordre partiel) entre les séquences d'entrée et de sortie du mécanisme d'imposition.

Remarque 2 (Imposition à l'exécution vs théorie de la supervision) *L'imposition à l'exécution partage ses objectifs avec la théorie du contrôle, introduite par Ramadge et Wonham [43, 44]. En théorie du contrôle, un automate modélisant le système est utilisé pour synthétiser un* superviseur *et une liste d'états interdits. Les événement du système sont partitionnés en événement contrôlables et événement incontrôlables. Intuitivement, le superviseur est composé avec l'automate modélisant le système (par produit synchrone) et assure le comportement le plus permissif du système tout en empêchant l'automate d'atteindre un mauvais état. Si le système essaye d'exécuter une action emmenant le système vers un mauvais état, le superviseur désactive cette action et celle-ci ne peut plus s'exécuter sur le système. Définir une approche d'imposition du système ne nécessite pas de modèle du système mais requiert uniquement une propriété exprimant un des bons comportements attendu de celui-ci. Les mécanismes d'imposition s'exécutent généralement en parallèle du système cible et ne sont pas intégrés dans celui-ci.*

2. Ceci est la notion de transparence adoptée dans une majorité d'articles sur l'imposition à l'exécution. Des travaux ont souligné que cette notion de transparences contraint uniquement les séquences d'exécution correctes ; et suggèrent que des contraintes devraient être formulées sur la manière dont un mécanisme transforme les séquences d'exécution incorrectes [6, 7, 27].

3 Imposition de propriétés avec temps logique

Cette section se consacre aux approches visant à imposer des propriétés avec temps logique. Les propriétés contraignent uniquement l'ordre dans lequel les actions doivent se produire. Dans la section 4, nous décrirons les approches d'imposition dédiées aux propriétés temporisées.

La section 3.1 donne une vue d'ensemble des principaux modèles de mécanismes utilisés lors de l'imposition à l'exécution. Nous verrons que ces modèles différent dans leur capacité d'imposer des propriétés et les hypothèses faites sur le système cible. La section 3.2 s'intéresse à la notion d'imposabilité de propriétés, c'est-à-dire les conditions permettant de déterminer s'il est possible d'imposer une propriété. La section 3.3 présente les travaux liés à la synthèse de mécanismes d'imposition.

3.1 Modèles de mécanismes d'imposition

Les *security automata* (SA) [49] sont le premier modèle de mécanisme d'imposition. Un SA est une machine à états finis qui s'exécute en parallèle du système surveillé. Dès que le système surveillé souhaite exécuter une action dans le champ de la propriété imposée, deux cas se présentent. Soit la transition est définie et alors le SA laisse le système cible exécuter l'action, soit le système est arrêté.

Les security automata sont (seulement) des *reconnaisseurs* de séquences. Ligatti et al. proposent le modèle des *edit-automata* (EAs) [31] qui sont des *transformateurs* de séquences. En plus d'arrêter le système cible, les edit-automata peuvent insérer et supprimer des actions (provenant du système cible ou pas). Par exemple, un EA peut supprimer et mémoriser une action pour éventuellement la rejouer plus tard. Dans un EA, la mémorisation des actions est réalisée directement en utilisant l'espace d'états (généralement infini) ; ceci par création d'un état pour les (groupe de) séquences d'entrées (équivalentes). Plusieurs variantes de edit-automata ont été proposées [6].

Falcone et al. généralisent les edit-automata automata avec les *generalised enforcement monitors* (GEMs) [22]. Contrairement aux EAs, un GEM sépare clairement la reconnaissance de la transformation de la séquence : les GEMs sont basés sur des machines à états finis étendues avec des opérations d'imposition génériques qui agissent sur la mémoire interne du mécanisme d'imposition. Séparer la reconnaissance de la séquence de la mémorisation a plusieurs avantages. D'abord, les GEMs sont plus facilement implémentables. Ensuite, il est facile de définir formellement des opérations de composition entre GEMs en calculant l'espace d'états produit et en composant les opérations sur la mémoire.

Bielova et Massacci ont proposé les *iterative suppression automata* (ISAs) [8], comme variante des EAs. Ils ont remarqué que les exigences usuelles de correction et transparence, et la manière dont celles-ci sont implémentées avec les EAs, ne permettent pas de déterminer quoi faire lorsque l'entrée ne satisfait pas la spécification. La motivation sous-jacente est de pouvoir comparer les EAs entre eux dans leur manière d'intervenir sur les exécutions incorrectes. Comme remarqué dans [15], les EAs et GEMs souffrent de limitations pratiques. Ces deux modèles supposent pouvoir suspendre un nombre non borné d'actions pour les rejouer plus tard. Ceci revient à supposer qu'un mécanisme d'imposition est capable de prédire le résultat de n'importe quelle action (sans l'exécuter) et transmettrait au système cible le résultat de cette action pour que le système continue à s'exécuter.

Pour résoudre le problème des EAs et GEMs précédemment mentionné, Dolzhenko et al. introduisent les Mandatory Results Automata (MRAs) [13, 33]. Lors de l'observation de toute action, un MRA doit renvoyer un résultat au système cible avant de pouvoir traiter l'action suivante dans la séquence. Un MRA est placé entre l'application cible non fiable et le système d'exécution. Il impose la propriété à la fois sur les actions exécutées par la cible et aussi sur le résultat de ses actions. Ainsi, un MRA doit considérer les événements d'entrée et de sortie dans les exécutions observées.

Dans [12,18], Charafeddine et al. proposent un mécanisme d'imposition avec des capacités de *roll-back* à k pas. Un tel mécanisme d'imposition autorise le système à dévier de la propriété jusqu'à k pas d'exécution observables. Si le système ne revient pas à un état correct après k pas, le mécanisme d'imposition restaure le système (non-déterministe) à son dernier état correct et le force à explorer les exécutions alternatives possiblement existantes. Une instanciation à 1 pas de cette définition générale de mécanismes d'imposition est implémentée et intégrée dans les systèmes à composants (cf. [3]).

Similaires aux modèles précédents sont les *safety shields* [9] pour les systèmes matériels réactifs, c'est-à-dire les systèmes avec signaux booléens pour les entrées et sorties. Un shield est une machine de Mealy qui assure l'exigence de correction et une interférence minimale. L'interférence est déterminée par une distance mesurant la déviation entre la sortie et l'entrée du shield. Lorsqu'un état violant la propriété devient inévitable, le shield entre dans une période de *recovery*, appelée *k-stabilisation*, et il est autorisé à dévier du comportement attendu pendant k pas consécutifs. Bloem et al. supposent ici que la violation est un événement rare et que le mécanisme traque

toutes les possibilités en supposant que l'erreur était isolée. Si une autre violation survient durant cette période de récupération, le shield entre dans un mode *fail-safe*, où la correction est toujours assurée mais aucune minimilalité n'est garantie pour la déviation. Notons qu'un shield ne peut pas mettre d'événements en mémoire tampon. Wu et al. étendent les shields et proposent des mécanismes d'imposition qui répondent immédiatement aux violations et garantissent le bon fonctionnement en cas de rafales d'erreurs [52].

Modèles avec contraintes mémoires. La plupart des modèles précédents de mécanismes d'imposition sont pourvus d'une mémoire infinie car ils offrent la possibilité de mémoriser un nombre non borné d'événements. Plusieurs modèles ont été proposés pour prendre en compte des limitations mémoire pouvant survenir en pratique et limitent la mémoire à disposition des mécanismes d'imposition. Fong propose les *shallow history automata* (SHAs) [23] comme security automata qui ne peuvent pas utiliser l'ordre d'arrivée des événements. Fong généralise la notion de SHA en α-SA qui sont des SA qui abstraient la séquence d'entrée courante selon un morphisme α. Talhi et al. introduisent les *bounded security automata* (BSAs) et les *bounded edit-automata* (BEAs) [50]. Les BSAs et les BEAs sont des SAs et des EAs avec une mémoire bornée pour mémoriser la séquence d'entrée, respectivement. Les précédents modèles bornent la taille de la mémoire des mécanismes d'imposition (avec un entier). Beauquier et al. introduisent les EAs finis et les EAs hors-contexte et déterministes, c'est-à-dire des EAs avec un ensemble fini d'états [5]. Ils prouvent que les EAs finis sont strictement moins expressifs que les EAs et étudient les conditions permettant d'imposer une propriété avec un EA fini.

Modèles prenant en compte les événements incontrôlables. Proches des contrôleurs en théorie de la supervision (voir Remarque 2), les mécanismes d'imposition prenant en compte les actions incontrolables (c'est-à-dire les actions ne pouvant être modifiées par un mécanisme d'imposition) ont été définies [26,46]. En plus de la satisfaction de la séquence de sortie, de tels modèles prennent en compte les réceptions possibles d'actions incontrôlables. Un premier type d'actions incontrôlables comme les ticks d'horloge ont été introduits par Basin et al. dans [2]. Des actions incontrôlables non restreintes ont été ensuite introduites dans des extensions des GEMs dans [45–47] et des EAs dans [26].

Mécanismes d'imposition prédictifs. Inspiré par le cadre de vérification à l'exécution avec sémantique prédictive [53], les mécanismes d'imposition prédictifs ont été proposés dans [41, 42]. Les mécanismes d'imposition prédictifs exploitent une connaissance statique du système (telle que les événements ou actions que le système peut produire ou va certainement produire dans le futur) pour produire des événements plus rapidement, au lieu de les retarder jusqu'à l'observation d'événements futurs.

3.2 Propriétés imposables

Nous nous intéressons maintenant aux caractérisations existantes des propriétés imposables, c'est-à-dire les ensembles de propriétés qui peuvent être imposées sur les systèmes cibles. Le fait qu'une spécification soit imposable dépend de plusieurs facteurs :

— le formalisme utilisé pour définir la propriété, et plus particulièrement le fait que le formalisme décrit des exécutions finies ou infinies ;
— les primitives d'imposition fournies au mécanisme d'imposition et comment des primitives sont associées à des actionneurs réels sur le système ;
— les contraintes venant du système dans lequel le mécanisme est placé.

Dans le travail pionnier de Schneider sur les security automata, les propriétés de sûreté ont été caractérisées comme imposables [49]. Comme un security automaton peut (seulement) soit laisser une action du système s'exécuter, soit arrêter le système définitivement, ses décisions sont irrémédiables. De manière concurrente, Kim et al. ont remarqué que n'importe quel mécanisme de surveillance (évaluant une exécution par rapport à une propriété) doit pouvoir déterminer si l'exécution courante est une exécution autorisée [29]. Ainsi, les propriétés doivent être *co-récursivement énumérables*, c'est-à-dire, le test de non-appartenance à l'ensemble des exécutions autorisées doit être calculable. Notons que dans [24], Schneider et al. corrigent et étendent les résultats dans [49] reliés aux capacités d'imposition des security automata (voir section 3.2) ; grâce aux résultats de [29].

Ligatti et al. ont prouvé que, en comparaison des security automata, en utilisant les primitives d'imposition additionnelles de suppression et d'insertion, les edit-automata peuvent imposer les propriétés dites de *renewal* [4, 31, 32]. Dans la classification safety-liveness des propriétés [36], les propriétés de renewal forment un sur-ensemble des propriétés de safety qui contient quelques propriétés de liveness. Intuitivement, une propriété est une renewal si a) n'importe quelle séquence d'exécution infinie satisfaisant la propriété contient un nombre infini

de préfixes dans la propriété et b) toute séquence infinie hors de la propriété contient seulement un nombre fini de préfixes dans la propriété.

Falcone et al. ont prouvé que les GEMs instanciés avec les opérations de mémorisation et de relâche d'événements pouvaient imposer des propriétés dîtes de *response* [16] dans la hiérarchie Safety-Progress [11]. Les propriétés de response sont des propriétés pour lesquelles un comportement désiré doit se produire un nombre infini de fois. Elles peuvent être comprises intuitivement comme des propriétés de transaction. De plus, nous notons que sur les séquences finies, toutes les propriétés sont des renewals. Cette observation colle avec le fait que les propriétés de (pure) response coincident avec les propriétés de renewal, comme remarqué dans [16].

Ligatti et al. ont prouvé que l'approche d'imposition avec MRA permet d'imposer une nouvelle sorte de propriétés, appelées propriétés de *result-sanitization* ou propriétés *monitor-centric* [13, 33]. De telles propriétés sont plus simples et plus expressives que les propriétés classiques centrés sur le systèmes cibles. Ils fournissent également une caractérisation sous forme de hiérarchie des propriétés imposables avec des MRAs. Par exemple, ils montrent que les MRAs imposent un sous-ensemble des propriétés de safety, alors que les Non-deterministic MRAs (NMRAs) imposent un sur-ensemble strict des propriétés de safety.

Falcone et Jaber [12, 18] ont montré que les propriétés de safety dites stutter-free [51] sont imposables sur les systèmes à composants avec des mécanismes d'imposition qui peuvent rétablir le système à son état précédent la dernière action observable. La stutter-invariance [51] est requise sur les propriétés à cause des contraintes venant des synchronisations ayant lieu entre composants. Ils proposent une hiérarchie de propriétés imposables en fonction du nombre d'états de l'historique que le système peut mémoriser (la notion dite d'imposabilité à k-step) [18].

Basin et al. étendent la caractérisation donnée dans [24,49] des propriétés imposables en considérant également un univers des exécutions d'entrée possibles et un ensemble d'actions contrôlables [2]. Une propriété est imposable si elle est de safety et est telle que les violations ne sont pas causées par les actions incontrôlables, et l'ensemble des préfixes des séquences et la propriété sont décidables.

3.3 Synthèse de mécanismes d'imposition

Nous présentons maintenant quelques travaux permettant de synthétiser des mécanismes d'imposition à partir de propriétés spécifiées dans plusieurs langages de spécification.

Schneider et al. synthétisent des SAs depuis des automates de Büchi. Ligatti et al. synthétisent des EAs depuis des automates à états finis décrivant des propriétés de renewal. Falcone et al. synthétisent des GEMs depuis les automates de Streett. Chabot et al. synthétisent des EAs depuis des automates de Rabin [10]. En utilisant des techniques de model-checking partiel, Mateucci et Martinelli synthétisent des SAs depuis des formules de μ-calcul [34]. Les mécanismes d'imposition sont décrits par des opérateurs algébriques et pilotés par des programmes contrôleurs. Falcone et Marchand synthétisent des GEMs depuis des systèmes de transition marqués avec des états secrets pour imposer l'opacité de propriétés sur le système [20]. Charafedine et al. transforment des automates à états finis déterministes en mécanismes d'imposition avec des capacités de restauration à 1-step et les intègrent dans des systèmes à composants. Bloem et al. synthétisent des safety shields depuis des automates de safety en résolvant un jeu à deux joueurs [9]. Wu et al. synthétisent des shields qui peuvent faire face à des rafales d'erreurs en utilisant un algorithme basé sur la théorie des jeux [52]. Bielova et Massacci adaptent la construction de EAs pour synthétiser une variante appelée *iteration suppression automata* pour les propriétés itératives décrites par des automates déterministes à états finis. Les propriétés itératives sont telles que les bonnes exécutions sont formées d'« itérations » qui peuvent être répétées un nombre arbitraire de fois.

4 Imposition de propriétés temporisées

Dans cette section, nous présentons les approches pour l'imposition de propriétés temporisées. Nous commençons par introduire le contexte spécifique aux approches d'imposition de propriétés temporisées ainsi que les problèmes soulevés dans ce contexte dans la section 4.1. Ensuite, nous donnons un historique de ces approches dans la section 4.2. Puis, dans la section 4.3, nous détaillons une succession d'approches pour les mécanismes d'imposition avec les possibilités d'imposition les plus puissantes permettant d'imposer les propriétés du temps continu. Enfin, dans la section 4.4, nous donnons un exemple détaillé illustrant le fonctionnement d'un mécanisme d'imposition pour une propriété du temps continu.

4.1 Contexte de l'imposition de propriétés temporisées

Les modèles de mécanismes d'imposition présentés dans la section 3 utilisent des procédures de reconnaissance de séquences et des primitives d'imposition non temporisées. En particulier, ces modèles ne prennent pas en compte le temps qui s'écoule entre deux événements ou actions. De plus, le temps pendant lequel un événement reste dans la mémoire du mécanisme d'imposition n'influence pas le comportement du mécanisme d'imposition ni la correction de la séquence produite par le mécanisme d'imposition. Les modèles de mécanismes d'imposition pour les propriétés temporisées doivent dans l'idéal prendre en compte le temps physique qui s'écoule durant la réception et la mémorisation d'événements ; ces primitives d'imposition doivent se faire en temps réel. Dans les modèles pour propriétés temporisées, le temps physique a des conséquences sur l'implémentabilité des mécanismes d'imposition. Ainsi, définir de tels modèles nécessite des notions de correction et transparence adaptées. De plus, des exigences additionnelles sont nécessaires pour déterminer la qualité des mécanismes d'imposition et comparer différentes versions de mécanismes d'imposition qui différeraient par la rapidité avec laquelle ils peuvent assurer la propriété désirée.

4.2 Historique des approches d'imposition de propriétés temporisées

L'idée d'utiliser des mécanismes d'imposition pour des propriétés temporisées a été premièrement discuté par Rinard dans [48]. Rinard étudie les différentes stratégies possibles pour surveiller et imposer des propriétés sur un système temps-réel. En particulier, Rinard suggère l'idée qu'un mécanisme d'imposition peut être utilisé pour retarder des événements pris en isolation dans un flux d'entrée lorsque ceux-ci arrivent trop tôt pour le système les traitant. L'étude menée dans [48] reste à un haut niveau d'abstraction et ne propose pas de description détaillée sur la manière d'obtenir de tels mécanismes d'imposition.

Plus tard, Matteucci s'inspire de techniques de model-checking partiel pour synthétiser des contrôleurs qui peuvent imposer des propriétés de safety et de flux d'information [35]. L'approche vise à imposer des propriétés sur le temps discret. Les propriétés sont modélisées par des processus temporisés exprimés en CCS. Les mécanismes d'imposition définis sont proches des security automata et des contrôleurs utilisés en théorie du contrôle. Ceux-ci sont intégrés dans le système et restreignent donc directement son comportement.

Assez récemment, Basin et al. ont proposé une approche pour l'imposition de propriétés de safety avec événements contrôlables et incontrôlables. Dans ce cadre, les événements incontrôlables servent à modéliser le temps : un tick d'horloge est un événement qu'un mécanisme d'imposition ne peut modifier. L'approche se focalise donc sur le temps discret et s'intéresse aux questions d'imposabilité des spécification et de synthèse de mécanismes d'imposition similaires aux security automata (qui arrêtent le système en cas d'erreur) à partir d'automates ou de formules logiques. Une ligne d'approches s'est par la suite intéressée à imposer des propriétés du temps continu pour des propriétés régulières spécifiée par des automates temporisés. Nous détaillons celles-ci dans la sous-section suivante.

4.3 Approches d'imposition de propriétés avec temps continu

Dans cette succession d'approches, suivant le papier séminal de Rinard, les mécanismes d'imposition sont des *retardateurs* s'exécutant en parallèle du système. Ces mécanismes d'imposition décident de la terminaison du système que lorsqu'il n'est pas possible de corriger la séquence d'entrée par retardement. Nous discutons les variantes de ces mécanismes d'imposition puis ensuite l'adaptation des exigences classiques de correction et transparence dans le cadre temporisé.

Mécanismes d'imposition pour les propriétés temporisées Pinisetty et al. ont introduit une première version de mécanismes d'imposition comme retardateur pour des propriétés de safety et co-safety [40]. L'approche a ensuite été étendue pour que ces mêmes mécanismes d'imposition puissent imposer n'importe quelle propriété régulière spécifiée par un automate temporisé [38]. L'extension aux propriétés paramétriques temporisées a été proposée dans [37]. Les propriétés paramétriques utilisent des événements augmentés de données provenant de l'exécution du système cible. Notons que cette dernière approche donne plusieurs exemples d'applications concrètes où l'imposition de propriétés paramétriques peut être utile comme dans les communications réseaux ou l'ordonnancement de processus. La possibilité pour un mécanisme d'imposition de supprimer des événements a été ensuite introduite dans [19] par Falcone et al. Ces mécanismes d'imposition sont des *retardateurs avec capacité de suppression*. Ces mécanismes d'imposition suppriment définitivement le dernier événement reçu lorsque celui-ci empêche la satisfaction de la propriété même après retardement. Des mécanismes d'imposition de type retardateur et qui prennent en compte les actions incontrôlables du système ont été introduits dans [46] puis optimisés dans [45] par Renard et al.

Exigences sur les mécanismes d'imposition Définir des mécanismes d'imposition pour des propriétés temporisées nécessite de redéfinir les exigences usuelles de correction et transparence (voir section 2). D'abord, bien évidemment, ces exigences doivent faire apparaitre explicitement le temps physique dans leur formulation. L'exigence de correction est généralement (nécessairement) affaiblie pour requérir que la séquence produite par le mécanisme d'imposition ne soit correcte qu'à partir d'un instant et continuellement à partir de cet instant. En effet, la propriété à imposer étant temporisée, le mécanisme d'imposition ne peut pas garantir que sa sortie soit correcte dès l'initialisation. L'exigence de transparence est également adaptée : la relation (d'ordre partiel ou d'équivalence) entre les séquences d'entrée et de sortie doit prendre en compte le temps ; avec des relations de préfixes ou de sous-séquences (cas des retardateurs avec suppression) entre séquences prenant en compte le temps. La séquence de sortie doit être généralement une version retardée de la séquence d'entrée (avec les mêmes actions dans le même ordre mais avec des instants du temps associés à ces actions plus grands). De plus, pour obtenir des descriptions de mécanismes d'imposition précises, il est nécessaire d'inclure des contraintes physiques modélisant le fait que les séquences d'entrée et de sortie sont des flux d'actions. Ces contraintes physiques modélisent par exemple le fait que les actions sont reçues de manière incrémentale au cours du temps ou qu'un événement ne peut être produit en sortie avant d'avoir été reçu. Aussi, une contrainte d'optimalité est ajoutée aux mécanismes d'imposition. Comme les mécanismes d'imposition sont des retardateurs, il est utile de pouvoir obtenir une notion d'optimalité décrivant le « meilleur » mécanisme d'imposition qui produit ses séquences de sortie le plus rapidement possible (tout en préservant la correction et transparence). Enfin, notons que dans le cas où le mécanisme d'imposition prend en compte les actions incontrôlables, il n'est plus possible de garantir la transparence : la réception d'actions incontrôlables en entrée (qui sont produites immédiatement en sortie) fait que l'ordre des actions n'est plus respecté ; les actions incontrôlables étant produites en sortie avant des actions contrôlables qui pourrait avoir été reçues auparavant et mise en mémoire du mécanisme d'imposition.

4.4 Exemple d'imposition d'une propriété avec temps continu

Nous développons maintenant un exemple illustrant le comportement des mécanismes d'imposition. La propriété considérée est une version étendue et adaptée d'une propriété de [45]. Nous considérons un mécanisme d'imposition temporisé de type retardateur avec capacité de suppression et prenant en compte les actions incontrôlable, combinant ainsi les capacités des mécanismes d'imposition définis dans [19] et [45].

Nous considérons une ressource accédée par un utilisateur qui peut faire des opérations de lecture et d'écriture sur cette ressource. La ressource est également administrée. L'administrateur de cette ressource peut décider une opération de maintenance. La maintenance est plus prioritaire que les opérations utilisateurs et celle-ci doit pouvoir avoir lieu dès que l'administrateur le décide. Ainsi, les opérations de lecture, d'écriture et d'effacement de l'utilisateur sont associées aux actions contrôlables `Lire`, `Ecrire` et `Efface` respectivement. La maintenance de l'administrateur est une opération longue qui est associée aux actions incontrôlables de début et de fin de maintenance *DebM* et *FinM* respectivement. Les actions incontrôlables sont mises en évidence en italique. Nous dénotons l'ensemble de toutes les actions par Σ.

Nous souhaitons ajouter à la ressource un mécanisme d'imposition permettant de réguler l'accès de l'utilisateur à la ressource de manière à ne pas interférer avec la maintenance de l'administrateur. Durant la maintenance, les lectures vers la ressource sont autorisées mais les écritures ne le sont pas. De plus, nous souhaitons assurer qu'au moins deux unités de temps s'écoulent entre la fin de la maintenance et les opérations d'écriture. Il n'y a pas de temps minimum requis ni entre les lectures ni entre les écritures. Enfin, nous souhaitons rendre impossible l'effacement (du contenu) de la ressource par l'utilisateur.

Pour cela, nous utilisons un automate temporisé décrivant les exécutions d'actions autorisées. Cet automate est représenté dans la Figure 2. Cet automate a trois localités l_1, l_2 (acceptantes) et l_3 (non acceptante) ainsi qu'une horloge x. L'horloge est re-initialisée lors de l'exécution de l'action marquant la fin de la maintenance (*FinM*) et sert à garder les opérations d'écriture lorsque la ressource n'est pas en maintenance.

Nous donnons une exemple d'imposition de cette propriété au cours du temps si la séquence d'événements (actions temporisées) reçue en entrée est $\sigma = (1, \texttt{Efface}).(2, \textit{DebM}).(4, \texttt{Ecrire}).(5, \textit{FinM}).(6, \textit{DebM}).(7, \texttt{Ecrire}).(8, \textit{FinM})$ où (t, a) désigne l'événement comportant l'action a et ayant lieu à t unités de temps après l'initialisation. Observons que cette exécution est incorrecte par rapport à la propriété. Nous étudions les configurations du mécanisme d'imposition, représentant son exécution. Nous représentons la configuration du mécanisme d'imposition par trois séquences σ_s, σ_b et σ_c, stockées dans les buffers du mécanisme d'imposition. (La configuration réelle d'une description opérationnelle d'un mécanisme d'imposition doit de plus comporter : 1) l'état atteint dans la sémantique de l'automate temporisée après avoir lu ce qui a été produit en sortie, 2) une horloge suivant le temps

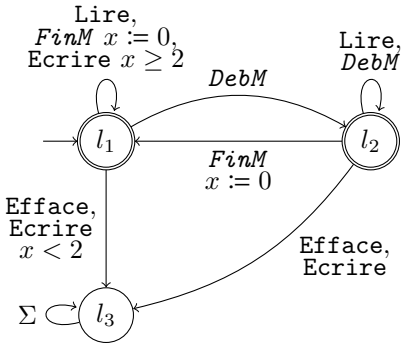


FIGURE 2: Propriété décrivant la bonne utilisation d'une ressource

TABLE 1: Valeurs de σ_s, σ_b et σ_c représentant les configurations du mécanisme d'imposition au cours du temps lors de la réception de la séquence $(1, \texttt{Efface}) \,.\, (2, \textit{DebM}) \,.\, (3, \texttt{Lire}) \,.\, (3, \texttt{Efface}) \,.\, (4, \texttt{Ecrire}) \,.\, (5, \textit{FinM}) \,.\, (6, \textit{DebM}) \,.\, (7, \texttt{Ecrire}) \,.\, (8, \textit{FinM})$ en entrée.

t	σ_s	σ_b	σ_c
0	ϵ	ϵ	ϵ
1	ϵ	ϵ	ϵ
2	$(2, \textit{DebM})$	ϵ	ϵ
3	$(2, \textit{DebM}) \,.\, (3, \texttt{Lire})$	ϵ	ϵ
4	$(2, \textit{DebM}) \,.\, (3, \texttt{Lire})$	ϵ	Ecrire
5	$(2, \textit{DebM}) \,.\, (3, \texttt{Lire}) \,.\, (5, \textit{FinM})$	$(7, \texttt{Ecrire})$	ϵ
6	$(2, \textit{DebM}) \,.\, (3, \texttt{Lire}) \,.\, (5, \textit{FinM}) \,.\, (6, \textit{DebM})$	ϵ	Ecrire
7	$(2, \textit{DebM}) \,.\, (3, \texttt{Lire}) \,.\, (5, \textit{FinM}) \,.\, (6, \textit{DebM})$	ϵ	Ecrire.Ecrire
8	$(2, \textit{DebM}) \,.\, (3, \texttt{Lire}) \,.\, (5, \textit{FinM}) \,.\, (6, \textit{DebM}) \,.\, (8, \textit{FinM})$	$(10, \texttt{Ecrire}) \,.\, (10, \texttt{Ecrire})$	ϵ
10	$(2, \textit{DebM}) \,.\, (3, \texttt{Lire}) \,.\, (5, \textit{FinM}) \,.\, (6, \textit{DebM}) \,.\, (8, \textit{FinM}) \,.\, (10, \texttt{Ecrire}) \,.\, (10, \texttt{Ecrire})$	ϵ	ϵ

qui s'écoule depuis l'initialisation, et 3) deux horloges re-initialisées à chaque réception et émission d'événement afin d'optimiser les calculs et faciliter l'implémentation.)

Les valeurs prises par σ_s, σ_b et σ_c au cours du temps sont données dans la Table 1. La séquence (temporisée) σ_s est la sortie du mécanisme d'imposition après t unité(s) de temps. La séquence (temporisée) σ_b est composée d'événements contrôlables qui doivent être relâchés par le mécanisme d'imposition après la date du dernier événement reçu en entrée, si aucun événement incontrôlable n'est reçu. La séquence (non temporisée) σ_c est composée des actions contrôlables restantes dans le buffer (qui ne peuvent être relâchées pour le moment). Lors de la réception d'un nouvel événement, le mécanisme d'imposition détermine d'abord si l'action de cet événement doit être supprimée ou non. Une action est (irrémédiablement) supprimée (et non mise dans buffer) si quelque soit la futur séquence d'entrée (possiblement corrigée), ajouter un événement avec cette action empêche de satisfaire la propriété ; c'est par exemple le cas avec l'action `Efface`. Nous décrivons maintenant l'évolution interdépendante de σ_s, σ_b et σ_c en cas de réception d'actions qui ne doivent pas être supprimées. Lorsque le temps s'écoule, après le dernier événement reçu en entrée, σ_s est modifiée pour produire en sortie les événements dans σ_b lorsque leur date d'émission est arrivée. Lors de la réception d'un nouvel événement en entrée, celui-ci est ajouté à σ_s si c'est un événement incontrôlable ou ajouté au buffer si c'est un événement contrôlable (qui ne doit pas être supprimé). Ensuite, σ_b est recalculée, en prenant en compte le nouvel état atteint dans (la sémantique de) l'automate si la précédente action était incontrôlable ou avec le nouveau buffer si c'était une action contrôlable.

Nous commentons les configurations du mécanisme d'imposition.

— À $t = 0$, les trois buffers σ_s, σ_b et σ_c sont vides.

— À $t = 1$, la réception de l'événement $(1, \texttt{Efface})$ mène à un état accepteur. De plus, quelque soit la future séquence reçue en entrée, il n'est pas possible de produire un événement avec l'action `Efface` et atteindre un état accepteur (toute mot contenant un événement `Efface` mène à un état non-accepteur). L'action `Efface` est donc supprimée et la configuration du mécanisme d'imposition n'est pas modifiée. (C'est également le cas lors de la réception de l'événement $(3, \texttt{Efface})$).

— À $t = 2$, lors de la réception de l'événement $(2, \texttt{DebM})$ avec l'action incontrôlable `DebM`, cet événement est directement placé dans σ_s. Le mécanisme d'imposition ne peut pas opérer sur cette action.
— À $t = 3$, lors de la réception de l'événement $(3, \texttt{Lire})$ avec l'action contrôlable `Lire`, comme cette action est autorisée durant la maintenance (après avoir atteint la localité l_2, après 1 unité de temps, la transition induite par l'action `Lire` permet de rester dans la localité acceptante l_2).
— À $t = 4$, lors de la réception de l'événement $(4, \texttt{Ecrire})$ avec l'action contrôlable `Ecrire`, la transition induite par cette action mènerait à la localité (non acceptante) l_3. Ainsi, cette action ne peut être placée dans σ_s et doit donc être bufferisée pour être possiblement produite en sortie plus tard. Notons que cette action ne doit pas être supprimée à cause de l'existence de chemins comprenant l'action `Ecrire` et menant à des localités acceptantes. De plus, comme il n'est pas possible de calculer une nouvelle date pour la production en sortie de cette action (car le fait de pouvoir produire cette action en sortie est conditionné par la fin de la maintenance - action incontrôlable `FinM`), cette action est placée dans le buffer σ_c.
— À $t = 5$, la réception de l'événement $(5, \texttt{FinM})$, avec l'action incontrôlable `FinM`, fait sortir de la maintenance. L'événement est directement placé dans le buffer σ_s pour être produit en sortie. De plus, la réception de cet événement permet de calculer une date pour l'action `Ecrire` précédemment reçu à $t = 4$. La date 7 est déterminée car elle correspond à 2 unités de temps après la date de l'émission de l'action `FinM` : c'est le temps minimal qui permet de produire en sortie une séquence correcte. Notons que l'événement $(7, \texttt{Ecrire})$ n'est pas placé dans σ_s mais dans σ_b car la décision de produire cet événement en sortie n'est pas définitive. En effet, la possibilité de pouvoir produire cet événement est conditionné par la non-réception d'événements incontrôlables jusqu'à la date d'émission de cette action (c'est-à-dire entre $t = 5$ et $t = 7$).
— À $t = 6$, la réception de l'événement $(6, \texttt{DebM})$, avec l'action incontrôlable `DebM`, fait entrer dans la maintenance. Cette action incontrôlable est placée directement dans le buffer σ_s avec l'événement $(6, \texttt{DebM})$. L'occurence de cet événement entraine que l'événement $(7, \texttt{Ecrire})$ (qui était placé dans le buffer σ_b) n'est plus valide (il ne peut plus être produit en sortie sans la réception d'un événement incontrôlable indiquant la fin de la maintenance). Ainsi, l'action `Ecrire` est placée dans le buffer σ_c.
— À $t = 7$, la réception de l'événement $(7, \texttt{Ecrire})$ implique similairement que l'action `Ecrire` soit placée dans le buffer σ_c car la fin de la maintenance est nécessaire avant de pouvoir calculer une date de sortie pour cette action (qui dépendra également de la date de sortie de l'action `Ecrire` déjà présente dans le buffer σ_c.
— À $t = 8$, la réception de l'événement $(8, \texttt{FinM})$, avec l'action incontrôlable `FinM`, fait sortir de la maintenance. L'événement est directement placé dans le buffer σ_s pour être produit en sortie. De plus, la réception de cet événement permet de calculer des dates pour les actions `Ecrire` précédemment reçues à $t = 4$ et $t = 7$. La date $t = 10$ est déterminée pour ces deux actions qui correspond à 2 unités de temps après l'occurence de la dernière action `FinM`. La deuxième action `Ecrire` sera également produite en sortie à $t = 10$ car il n'y a pas de temps minimum requis entre deux actions `Ecrire`. Ces deux événements sont placés dans le buffer σ_b pour être produits en sortie à $t = 10$ si aucun événement incontrôlable empêchant de produire ces événements n'est reçu.
— À $t = 10$, comme aucune action incontrôlable n'a été reçue, les deux événements $(10, \texttt{Ecrire})$ sont placés dans le buffer σ_s.

5 Conclusions

La société est en demande de systèmes dynamiques à grande échelle et à haute fiabilité qui peuvent servir leur citoyens. Ces systèmes sont toujours plus difficiles à vérifier à cause de leur taille et complexité toujours croissantes et de leur nature dynamique. Les techniques de vérification et d'imposition à l'exécution peuvent être exploitées lorsque les systèmes sont en production pour complémenter les activités de vérification et de validation réalisées durant le développement.

Cet article discute certains résultats dans le domaine de l'imposition à l'exécution. Le matériel présenté a pour but d'être un point de départ pour les chercheurs intéressés par développer des solutions d'imposition de propriétés.

Remerciements

Cet article est basé en partie sur i) des publications [19, 21, 38–40, 45, 46] réalisées avec Thierry Jéron, Hervé Marchand, Srinivas Pinisetty, Matthieu Renard et Antoine Rollet durant d'actives collaborations ces dernières

années, un chapitre de livre en cours de soumission (co-écrit avec Léonardo Mariani, Antoine Rollet et Saikat Saha) et également ii) une précédente publication de l'auteur [14].

L'auteur souhaite remercier Antoine El-Hokayem, Raphaël Jakse et Jean-François Mehaut pour leur relectures sur les versions préliminaires de cet article.

Références

1. Bartocci, E., Falcone, Y., Bonakdarpour, B., Colombo, C., Decker, N., Havelund, K., Joshi, Y., Klaedtke, F., Milewicz, R., Reger, G., Rosu, G., Signoles, J., Thoma, D., Zalinescu, E., Zhang, Y. : First international competition on runtime verification rules, benchmarks, tools, and final results of CSRV 2014. STTT (2017), to Appear
2. Basin, D., Jugé, V., Klaedtke, F., Zălinescu, E. : Enforceable security policies revisited. ACM Trans. Inf. Syst. Secur. 16(1), 3 :1–3 :26 (Jun 2013), http ://doi.acm.org/10.1145/2487222.2487225
3. Basu, A., Bensalem, S., Bozga, M., Combaz, J., Jaber, M., Nguyen, T., Sifakis, J. : Rigorous component-based system design using the BIP framework. IEEE Software 28(3), 41–48 (2011)
4. Bauer, L., Ligatti, J., Walker, D. : More enforceable security policies. In : Proceedings of the Workshop on Foundations of Computer Security (FCS'02), Copenhagen, Denmark (2002)
5. Beauquier, D., Cohen, J., Lanotte, R. : Security policies enforcement using finite and pushdown edit automata. Int. J. Inf. Sec. 12(4), 319–336 (2013), http ://dx.doi.org/10.1007/s10207-013-0195-8
6. Bielova, N., Massacci, F. : Do you really mean what you actually enforced ? - edited automata revisited. Int. J. Inf. Sec. 10(4), 239–254 (2011)
7. Bielova, N., Massacci, F. : Predictability of enforcement. In : Erlingsson, Ú., Wieringa, R., Zannone, N. (eds.) Engineering Secure Software and Systems - Third International Symposium, ESSoS 2011, Madrid, Spain, February 9-10, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6542, pp. 73–86. Springer (2011)
8. Bielova, N., Massacci, F. : Iterative enforcement by suppression : Towards practical enforcement theories. Journal of Computer Security 20(1), 51–79 (2012)
9. Bloem, R., Könighofer, B., Könighofer, R., Wang, C. : Shield synthesis : - runtime enforcement for reactive systems. In : Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings. pp. 533–548 (2015)
10. Chabot, H., Khoury, R., Tawbi, N. : Generating in-line monitors for rabin automata. In : Jøsang, A., Maseng, T., Knapskog, S.J. (eds.) Identity and Privacy in the Internet Age, 14th Nordic Conference on Secure IT Systems, NordSec 2009, Oslo, Norway, 14-16 October 2009. Proceedings. Lecture Notes in Computer Science, vol. 5838, pp. 287–301. Springer (2009), http ://dx.doi.org/10.1007/978-3-642-04766-4
11. Chang, E., Manna, Z., Pnueli, A. : The Safety-Progress Classification. Tech. rep., Stanford University, Dept. of Computer Science (1992)
12. Charafeddine, H., El-Harake, K., Falcone, Y., Jaber, M. : Runtime enforcement for component-based systems. In : Wainwright, R.L., Corchado, J.M., Bechini, A., Hong, J. (eds.) Proceedings of the 30th Annual ACM Symposium on Applied Computing, Salamanca, Spain, April 13-17, 2015. pp. 1789–1796. ACM (2015)
13. Dolzhenko, E., Ligatti, J., Reddy, S. : Modeling runtime enforcement with mandatory results automata. International Journal of Information Security 14(1), 47–60 (Feb 2015)
14. Falcone, Y. : You should better enforce than verify. In : Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G.J., Rosu, G., Sokolsky, O., Tillmann, N. (eds.) Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6418, pp. 89–105. Springer (2010)
15. Falcone, Y., Fernandez, J.C., Mounier, L. : Synthesizing enforcement monitors wrt. the safety-progress classification of properties. In : Sekar, R., Pujari, A. (eds.) Information Systems Security, Lecture Notes in Computer Science, vol. 5352, pp. 41–55. Springer Berlin Heidelberg (2008)
16. Falcone, Y., Fernandez, J.C., Mounier, L. : What can you verify and enforce at runtime ? International Journal on Software Tools for Technology Transfer 14(3), 349–382 (2012)
17. Falcone, Y., Havelund, K., Reger, G. : A tutorial on runtime verification. In : Broy, M., Peled, D.A., Kalus, G. (eds.) Engineering Dependable Software Systems, NATO Science for Peace and Security Series, D : Information and Communication Security, vol. 34, pp. 141–175. IOS Press (2013)
18. Falcone, Y., Jaber, M. : Fully automated runtime enforcement of component-based systems with formal and sound recovery. International Journal on Software Tools for Technology Transfer pp. 1–25 (2016)

19. Falcone, Y., Jéron, T., Marchand, H., Pinisetty, S. : Runtime enforcement of regular timed properties by suppressing and delaying events. Systems & Control Letters 123, 2–41 (2016)
20. Falcone, Y., Marchand, H. : Enforcement and validation (at runtime) of various notions of opacity. Discrete Event Dynamic Systems 25(4), 531–570 (2015), http ://dx.doi.org/10.1007/s10626-014-0196-4
21. Falcone, Y., Marchand, H. : Enforcement and validation (at runtime) of various notions of opacity. Discrete Event Dynamic Systems 25(4), 531–570 (2015)
22. Falcone, Y., Mounier, L., Fernandez, J., Richier, J. : Runtime enforcement monitors : composition, synthesis, and enforcement abilities. Formal Methods in System Design 38(3), 223–262 (2011)
23. Fong, P.W.L. : Access control by tracking shallow execution history. In : 2004 IEEE Symposium on Security and Privacy (S&P 2004), 9-12 May 2004, Berkeley, CA, USA. pp. 43–55. IEEE Computer Society (2004)
24. Hamlen, K.W., Morrisett, G., Schneider, F.B. : Computability classes for enforcement mechanisms. ACM Transactions on Programming Languages and Systems (TOPLAS) 28(1), 175–205 (2006)
25. Havelund, K., Goldberg, A. : Verify your runs. In : Meyer, B., Woodcock, J. (eds.) Verified Software : Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions. Lecture Notes in Computer Science, vol. 4171, pp. 374–383. Springer (2005)
26. Khoury, R., Hallé, S. : Runtime enforcement with partial control. In : García-Alfaro, J., Kranakis, E., Bonfante, G. (eds.) Foundations and Practice of Security - 8th International Symposium, FPS 2015, Clermont-Ferrand, France, October 26-28, 2015, Revised Selected Papers. Lecture Notes in Computer Science, vol. 9482, pp. 102–116. Springer (2015)
27. Khoury, R., Tawbi, N. : Corrective enforcement : A new paradigm of security policy enforcement by monitors. ACM Trans. Inf. Syst. Secur. 15(2), 10 :1–10 :27 (Jul 2012)
28. Khoury, R., Tawbi, N. : Which security policies are enforceable by runtime monitors ? A survey. Computer Science Review 6(1), 27–45 (2012)
29. Kim, M., Kannan, S., Lee, I., Sokolsky, O., Viswanathan, M. : Computational analysis of run-time monitoring - fundamentals of java-mac. Electr. Notes Theor. Comput. Sci. 70(4), 80–94 (2002)
30. Leucker, M., Schallhart, C. : A brief account of runtime verification. Journal of Logic and Algebraic Programming 78(5), 293–303 (may/june 2008)
31. Ligatti, J., Bauer, L., Walker, D. : Enforcing non-safety security policies with program monitors. In : di Vimercati, S.D.C., Syverson, P.F., Gollmann, D. (eds.) Computer Security - ESORICS 2005, 10th European Symposium on Research in Computer Security, Milan, Italy, September 12-14, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3679, pp. 355–373. Springer (2005)
32. Ligatti, J., Bauer, L., Walker, D. : Run-time enforcement of nonsafety policies. ACM Trans. Inf. Syst. Secur. 12(3), 19 :1–19 :41 (Jan 2009)
33. Ligatti, J., Reddy, S. : A theory of runtime enforcement, with results. In : Gritzalis, D., Preneel, B., Theoharidou, M. (eds.) Computer Security - ESORICS 2010, 15th European Symposium on Research in Computer Security, Athens, Greece, September 20-22, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6345, pp. 87–100. Springer (2010)
34. Martinelli, F., Matteucci, I. : Through modeling to synthesis of security automata. Electr. Notes Theor. Comput. Sci. 179, 31–46 (2007), http ://dx.doi.org/10.1016/j.entcs.2006.08.029
35. Matteucci, I. : Automated synthesis of enforcing mechanisms for security properties in a timed setting. Electronic Notes in Theoretical Computer Science 186, 101–120 (2007)
36. Owicki, S., Lamport, L. : Proving liveness properties of concurrent programs. ACM Transaction Programming Languages and Systems 4(3), 455–495 (1982)
37. Pinisetty, S., Falcone, Y., Jéron, T., Marchand, H. : Runtime enforcement of parametric timed properties with practical applications. In : Lesage, J., Faure, J., Cury, J.E.R., Lennartson, B. (eds.) 12th International Workshop on Discrete Event Systems, WODES 2014, Cachan, France, May 14-16, 2014. pp. 420–427. International Federation of Automatic Control (2014)
38. Pinisetty, S., Falcone, Y., Jéron, T., Marchand, H. : Runtime enforcement of regular timed properties. In : Cho, Y., Shin, S.Y., Kim, S., Hung, C., Hong, J. (eds.) Symposium on Applied Computing, SAC 2014, Gyeongju, Republic of Korea - March 24 - 28, 2014. pp. 1279–1286. ACM (2014)
39. Pinisetty, S., Falcone, Y., Jéron, T., Marchand, H. : Tipex : A tool chain for timed property enforcement during execution. In : Bartocci, E., Majumdar, R. (eds.) Runtime Verification - 6th International Conference, RV 2015 Vienna, Austria, September 22-25, 2015. Proceedings. Lecture Notes in Computer Science, vol. 9333, pp. 306–320. Springer (2015)
40. Pinisetty, S., Falcone, Y., Jéron, T., Marchand, H., Rollet, A., Nguena-Timo, O.L. : Runtime enforcement of timed properties. In : Qadeer, S., Tasiran, S. (eds.) Runtime Verification, Third International Conference, RV 2012, Istanbul, Turkey, September 25-28, 2012, Revised Selected Papers. Lecture Notes in Computer Science, vol. 7687, pp. 229–244. Springer (2012)

41. Pinisetty, S., Preoteasa, V., Tripakis, S., Jéron, T., Falcone, Y., Marchand, H. : Predictive runtime enforcement. In : Ossowski, S. (ed.) Proceedings of the 31st Annual ACM Symposium on Applied Computing, Pisa, Italy, April 4-8, 2016. pp. 1628–1633. ACM (2016)
42. Pinisetty, S., Preoteasa, V., Tripakis, S., Jéron, T., Falcone, Y., Marchand, H. : Predictive runtime enforcement. Formal Methods in System Design pp. 1–46 (2017)
43. Ramadge, P.J., Wonham, W.M. : Supervisory control of a class of discrete event processes. SIAM journal on control and optimization 25(1), 206–230 (1987)
44. Ramadge, P.J., Wonham, W.M. : The control of discrete event systems. Proceedings of the IEEE 77(1), 81–98 (1989)
45. Renard, M., Falcone, Y., Rollet, A., Jéron, T., Marchand, H. : Optimal enforcement of (timed) properties with uncontrollable events. Mathematical Structures in Computer Science p. 1–46 (2017)
46. Renard, M., Falcone, Y., Rollet, A., Pinisetty, S., Jéron, T., Marchand, H. : Enforcement of (timed) properties with uncontrollable events. In : Leucker, M., Rueda, C., Valencia, F.D. (eds.) Theoretical Aspects of Computing - ICTAC 2015 - 12th International Colloquium Cali, Colombia, October 29-31, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9399, pp. 542–560. Springer (2015)
47. Renard, M., Rollet, A., Falcone, Y. : Runtime enforcement using büchi games. In : Proceedings of Model Checking Software - 24th International Symposium, SPIN 2017, Co-located with ISSTA 2017, Santa Barbara, USA. pp. 70–79. ACM (July 2017), http ://www.labri.fr/perso/rollet/divers/spin17.pdf
48. Rinard, M. : Acceptability-oriented computing. In : Crocker, R., Jr., G.L.S. (eds.) Proceedings of the 2003 ACM SIGPLAN Conference On Object-Oriented Programming Sytems, Languages, and Applications Companion (OOPSLA 03 COMPANION). pp. 221–239. ACM Press (2003)
49. Schneider, F.B. : Enforceable security policies. ACM Trans. Inf. Syst. Secur. 3(1), 30–50 (Feb 2000)
50. Talhi, C., Tawbi, N., Debbabi, M. : Execution monitoring enforcement under memory-limitation constraints. Inf. Comput. 206(2-4), 158–184 (2008), http ://dx.doi.org/10.1016/j.ic.2007.07.009
51. Wilke, T. : Classifying discrete temporal properties. In : STACS. LNCS, vol. 1563, pp. 32–46. Springer (1999)
52. Wu, M., Zeng, H., Wang, C. : Synthesizing runtime enforcer of safety properties under burst error. In : Rayadurgam, S., Tkachuk, O. (eds.) NASA Formal Methods - 8th International Symposium, NFM 2016, Minneapolis, MN, USA, June 7-9, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9690, pp. 65–81. Springer (2016)
53. Zhang, X., Leucker, M., Dong, W. : Runtime verification with predictive semantics. In : Goodloe, A., Person, S. (eds.) NASA Formal Methods - 4th International Symposium, NFM 2012, Norfolk, VA, USA, April 3-5, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7226, pp. 418–432. Springer (2012)