Kairos : de la spécification temporelle de sûreté à la preuve locale

Frédéric Dabrowski

Kairos : de la spécification temporelle de sûreté à la preuve locale

Frédéric Dabrowski
Univ.Orléans, INSA Centre Val de Loire, LIFO EA 4022
Orléans, France
frederic.dabrowski@univ-orleans.fr

Résumé. Cet article introduit Kairos, un outil de vérification déductive pour une classe de programmes réactifs synchrones. L'exécution de ces programmes procède en pas successifs, appelés instants.À chaque instant, un programme consomme un élément de chaque flux d'entrée, exécute un calcul, puis produit un élément sur chaque flux de sortie. Son comportement est spécifié par des contrats LTL de sûreté dont les atomes sont des formules du premier ordre sur l'historique des flux d'entrée et de sortie. Cette spécification exprime les relations attendues entre les historiques des entrées et des sorties au cours des instants. Kairos utilise une réduction sémantique de ces contrats globaux en obligations locales de pas. Cette réduction repose sur la construction d'automates de sûreté et de produits synchronisés à partir desquels une représentation intermédiaire est utilisée pour la production des obligations locales. Ces obligations sont finalement traduites trivialement vers un backend de vérification déductive classique pour être déchargées. L'article décrit précisément cette chaîne sur un fil rouge.

Mots-clés. Systèmes synchrones, LTL, automates de sûreté, produits synchronisés, vérification déductive.

1 Introduction

Un système réactif synchrone peut être vu comme un transformateur de flux : à chaque cycle, il lit une valeur sur chaque entrée, exécute un pas de calcul, puis produit une valeur sur chaque sortie [1, 2]. La sémantique observable est une sémantique de mots infinis, où chaque position encode les valeurs d'entrée et de sortie associées à un pas d'exécution. Pour ces systèmes, la spécification s'exprime classiquement dans des logiques temporelles, interprétées sur les exécutions, dont les formes les plus connues sont LTL (traces individuelles) et CTL/CTL* (quantification explicite sur les chemins) [3, 4]. Dans le cadre de LTL, l'approche la plus classique pour vérifier la correction d'un programme traduit la formule en automate de Büchi sur mots infinis, puis ramène la vérification à un problème de reconnaissance de langage (et, en model checking, à un test d'emptiness sur le produit système $\times$ automate de la négation de la formule) [5, 6]. Si les approches de model checking offrent des garanties fortes, leur portée pratique reste souvent contrainte par l'explosion combinatoire de l'espace d'états et par la nécessité de restreindre les domaines de données pour préserver la décidabilité.

Le langage Kairos permet de programmer des systèmes réactifs synchrones, de spécifier leur comportement par des contrats temporels et d'en établir la correction au moyen de techniques de vérification déductive. Plus précisément, le comportement attendu d'un programme est décrit par une hypothèse (`requires`) sur ses entrées et une garantie sur ses sorties (`ensures`). Prouver la correction d'un programme consiste à montrer que pour toute exécution π (un mot infini) de ce programme, $\pi \models \texttt{requires} \Rightarrow \pi \models \texttt{ensures}$. Les contrats de Kairos sont des propriétés de *sûreté* exprimées en LTL. Ces propriétés, notées ϕ et ψ, sont construites à partir d'atomes (des formules du premier ordre sur l'historique), des formules $\texttt{X}\,\phi$, $\texttt{G}\,\phi$, $\phi\,\texttt{W}\,\psi$ et les opérations logiques usuelles (sous des contraintes garantissant l'appartenance au fragment de sûreté). Intuitivement, $\texttt{X}\,\phi$ impose ϕ au pas suivant, $\texttt{G}\,\phi$ impose ϕ à tous les pas, et $\phi\,\texttt{W}\,\psi$ impose que ϕ reste vraie jusqu'à ψ si ψ survient, sinon que ϕ reste vraie indéfiniment. Pour de telles propriétés, toute violation admet un témoin fini : il existe un préfixe *bad* de l'exécution qui suffit à certifier l'erreur [7, 8]. Dans ce cadre, des automates de sûreté suffisent à représenter la contrainte temporelle pertinente, sans mobiliser toute la machinerie des automates de Büchi pour la vivacité. La vérification se reformule alors comme une condition sur les préfixes (des mots finis) : aucun préfixe admissible ne doit conduire à une transition interdite.

Dans Kairos, la condition d'acceptation sur les automates est réduite à des conditions locales par pas : pour chaque pas exécuté sous un contexte admissible, il faut (i) établir une progression locale vers au moins un état admissible, et (ii) exclure les états interdits. L'automate n'est pas ici la cible finale de décision ; il sert de structure intermédiaire pour construire des obligations locales, tout en préservant l'interprétation temporelle globale. Cette lecture explicite l'articulation entre deux plans complémentaires : un plan temporel global (traces, automates, reconnaissance de mots) et un plan local (obligations de pas, preuves déductives).

Un programme Kairos est défini par un automate de contrôle dont chaque transition porte un fragment de code impératif. À chaque instant, une transition est sélectionnée en fonction des valeurs courantes des entrées et de l'état

interne (état de contrôle et variables locales) ; son exécution détermine l'état suivant et les valeurs de sortie. La chaîne opérationnelle de KAIROS construit les automates associés aux propriétés `requires`/`ensures`, leur produit synchronisé avec le contrôle du programme, puis une représentation intermédiaire (IR) qui explicite les obligations locales. Cet IR est ensuite déchargé vers un backend déductif classique, en l'occurrence Why3 [9]. L'approche n'est limitée, pour les constructions de programmation admises comme pour les théories de données traitables, que par le pouvoir expressif du backend déductif.

La contribution de l'article est de présenter cette réduction de manière opérationnelle et rigoureuse, au travers d'un fil rouge ; le prototype est disponible à l'adresse https://github.com/DabrowskiFr/Kairos.git, avec des exemples, des contre-exemples et la chaîne de reproduction des artefacts présentés dans ce document. Le reste du document est organisé comme suit. La Section 2 présente le fil rouge et une lecture informelle de sa sémantique. La Section 3 détaille la construction des automates et du produit synchronisé. La Section 4 décrit les transformations de l'IR phase par phase. La Section 5 rapporte les mesures expérimentales. La Section 6 discute la portée et les limites de l'approche, et la Section 7 situe le travail dans la littérature. La Section 8 synthétise les résultats et ouvre les perspectives.

2 Fil Rouge

Un programme Kairos est gouverné par un automate de contrôle qui, à chaque instant, sélectionne une transition activable, exécute son code et produit les sorties correspondantes. Les entrées sont renouvelées par l'environnement à chaque instant, tandis que les sorties et les variables locales constituent l'état transmis d'un instant au suivant. Ce langage est fortement inspiré du langage intermédiaire OBC (*Object-Based Code*) de LUSTRE ets des automates de modes [1, 10, 11]. Dans cette section, nous introduisons le fil rouge `resettable_delay` (Figure 1) et présentons une description informelle de sa sémantique et de sa spécification.

```
node resettable_delay(reset: int, x: int) returns (y: int)
contracts
  requires : G ((reset = 0 or reset = 1) and (reset = 1 => x = 0));
  ensures  : G ((reset = 1) => (y = 0));
  ensures  : X G ((reset = 0 and prev reset = 1) => y = 0);
  ensures  : X G ((reset = 0 and prev reset = 0) => y = prev x);
locals
  m : int;
states
  Init(init), Run;
invariants
  in Run:
    (prev reset = 1 => m = 0) and (prev reset = 0 => m = prev x);
transitions
  Init:
    to Run when reset = 1 { y := 0; m := 0; }
    to Run when reset = 0 { y := 0; m := x; }
  Run:
    to Run when reset = 1 { y := 0; m := 0; }
    to Run when reset = 0 { y := m; m := x; }
end
```

FIGURE 1 – Code source Kairos de `resettable_delay`.

Programme. Le nœud manipule deux entrées (`x`, `reset`), une sortie (`y`) et une variable locale de mémoire (`m`). Il implémente un délai d'un instant réinitialisable : en mode nominal (`reset=0`), la sortie publie la valeur mémorisée au pas précédent (0 au premier pas), puis la mémoire est mise à jour avec l'entrée courante ; en mode reset (`reset=1`), sortie et mémoire sont forcées à zéro. Son contrôle est donné par deux états, `Init` (état initial, marqué `init`) et `Run`. Chaque bloc de transition est défini par un état source et une liste ordonnée de clauses, chacune précisant une garde, un état destination et un corps impératif. Dans cet exemple, la transition `Init -> Run` initialise le régime permanent ; en `Run`, deux transitions `Run -> Run` coexistent : l'une active sous `reset=1` (remise à zéro), l'autre sous `reset=0` (mode nominal avec retard d'un instant). Lorsque plusieurs transitions sont activables, Kairos applique l'ordre de déclaration. À ce stade, le fragment impératif de KAIROS est volontairement minimal ; il est toutefois conçu pour être étendu à d'autres constructions de programmation et à des structures de données plus riches, dès lors qu'elles admettent une traduction correcte et exploitable par le backend de preuve.

Spécification. La spécification associe au nœud des clauses temporelles conditionnelles `requires`/`ensures` du fragment de sûreté de LTL, dont les atomes sont des formules du premier ordre sur l'historique des flux d'entrée et de sortie. Kairos utilise des termes de la forme `prev k x` ; sémantiquement, `prev k x` désigne la valeur de la variable x observée k pas en arrière (`prev x` correspondant à $k = 1$), lorsque celle-ci est définie. Les clauses `requires` fixent les hypothèses d'environnement, les clauses `ensures` expriment les obligations sur les sorties sous ces hypothèses. Les invariants de nœud, quant à eux, sont des assertions locales attachées à un état de contrôle

et requises à chaque instant où l'exécution démarre dans cet état. Le langage impose une restriction syntaxique sur ces invariants : ils ne peuvent pas référencer les valeurs d'entrée de l'instant courant. Sur ce fil rouge, ces deux niveaux sont complémentaires : le contrat exprime l'exigence temporelle globale, tandis que l'invariant de `Run` stabilise la relation entre mémoire et historique nécessaire aux preuves locales. La clause `requires` borne les entrées admissibles, les clauses `ensures` imposent que `y` respecte la remise à zéro et le délai d'un pas attendu. Comme pour les constructions impératives, les contrats temporels du fil rouge sont volontairement simples afin de rendre la chaîne de réduction lisible ; en pratique, le langage de spécification accepte des formules avec des imbrications plus complexes d'opérateurs temporels, tant que la formule reste une propriété de sûreté.

3 Automates d'hypothèses, de garanties et produit synchronisé

3.1 Hypothèses et garanties

Le contrat d'un noeud est traduit en deux automates de sûreté : un automate d'hypothèses $\mathcal{R}$ pour `requires` et un automate de garanties $\mathcal{E}$ pour `ensures`. La construction repose sur l'outil Spot[12] qui permet de rejetter les formules qui ne correspondent pas à des propriétés de sûreté. Elle suit un schéma standard de *proposification* : les atomes sont d'abord abstraits en propositions, Spot synthétise l'automate, puis les atomes sont réinjectés sur les arêtes. Les formules obtenues sur les transitions sont ensuite simplifiées par appel à Z3.

Dans la suite, nous notons ϕ^i les formules portées par les arêtes de $\mathcal{R}$, et ψ^i celles portées par les arêtes de $\mathcal{E}$. Les automates $\mathcal{R}$ et $\mathcal{E}$ générés pour notre exemple sont représentés à la Figure 2, tandis que les formules correspondant aux étiquettes des transitions sont données dans la Table 1. L'automate $\mathcal{R}$ comporte trois transitions : $\phi^2 = (reset = 1 \wedge x = 0) \vee (reset = 0)$, correspondant au cas admissible ; ϕ^1, définie comme son complément, correspondant au cas interdit ; et $\phi^3 = \top$, associée à la situation où un état interdit a déjà été atteint (absorbant, R^{bad}). L'automate $\mathcal{E}$ regroupe les trois clauses `ensures` du programme : ψ^1 et ψ^4 correspondent aux transitions admissibles, ψ^2 et ψ^3 correspondent aux transitions interdites, et $\psi^5 = \top$ (absorbant, E^{bad}). L'automate $\mathcal{E}$ se lit selon le même principe que $\mathcal{R}$, mais sa structure est plus riche car il encode la conjonction des trois clauses `ensures`.

3.2 Produit synchronisé

Un état du produit synchronisé est un triplet (S, R, E), où S est un état de contrôle du programme, R un état de l'automate $\mathcal{R}$, et E un état de l'automate $\mathcal{E}$. Le produit est construit par exploration des états accessibles, à partir de l'état initial formé de l'état initial du programme et des états initiaux de $\mathcal{R}$ et $\mathcal{E}$. Une transition $\tau = (g, \phi, \psi)$ relie deux états (S^1, R^1, E^1) et (S^2, R^2, E^2) du produit si et seulement si :

- (S^1, R^1, E^1) est accessible ;
- g est la garde d'une transition du programme de S^1 vers S^2 ;
- ϕ est l'étiquette d'une transition de R^1 vers R^2 dans $\mathcal{R}$;
- ψ est l'étiquette d'une transition de E^1 vers E^2 dans $\mathcal{E}$.

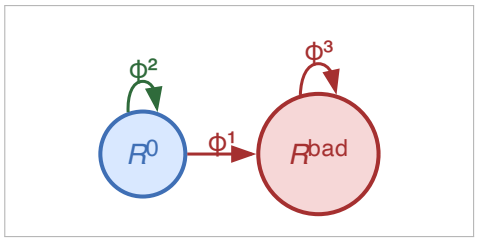
Lorsque plusieurs transitions programme partent d'un même état de contrôle, Kairos les priorise selon l'ordre de déclaration en renforçant chaque garde par la négation des gardes précédentes, ce qui évite les recouvrements dans la construction des obligations locales. La représentation des transitions diffère d'un produit synchronisé classique : Kairos conserve explicitement les triplets (g, ϕ, ψ), au lieu de les aplatir en une conjonction unique. Cette distinction est sémantiquement motivée : g et ϕ portent sur l'état au début du cycle (après rafraîchissement des entrées par l'environnement), tandis que ψ porte sur l'état en fin de cycle, après exécution du code. Les transitions du produit s'interprètent donc comme des contraintes sur des paires (σ, σ'), avec σ état d'entrée du pas et σ' état de sortie du pas. Sous cette lecture, l'automate produit reconnaît des mots sur l'alphabet des paires (σ, σ') vérifiant la contrainte de cohérence suivante : pour deux lettres successives (σ_i, σ'_i), $(\sigma_{i+1}, \sigma'_{i+1})$, l'état σ_{i+1} ne diffère de σ'_i que sur les entrées.

Le troisième schéma de la Figure 2 montre le produit de notre exemple ; les étiquettes sont données Table 1 sous la forme (g, ϕ, ψ). Les arêtes hachées correspondent aux transitions d'un état déjà contenant E^{bad} (trace déjà invalide), ou menant vers R_{bad} (entrée non valide) ; elles sont indicatives et ne sont pas utilisées ensuite. Les arêtes rouges pleines mènent vers un état contenant E^{bad} et capturent le moment où l'exécution produit un pas invalide. À titre d'exemple, depuis (Run, R^0, E^1), la transition programme `Run -> Run` sous $reset = 0$ produit deux transitions du produit : τ^5 (transition sûre sous la condition ψ^4) et τ^6 (transition non sûre sous la condition ψ^3).

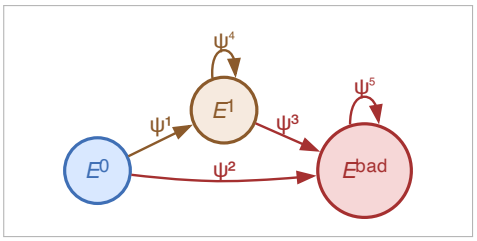
L'objectif de Kairos est double à chaque pas sur une entrée valide : garantir qu'au moins une transition sûre est réalisable (progression), et exclure toute transition non sûre. En pratique, les transitions sûres servent à la fois à établir la progression locale et à propager l'information temporelle d'un pas au suivant. Les transitions non sûres portent les conditions d'exclusion.

4 Représentation intermédiaire

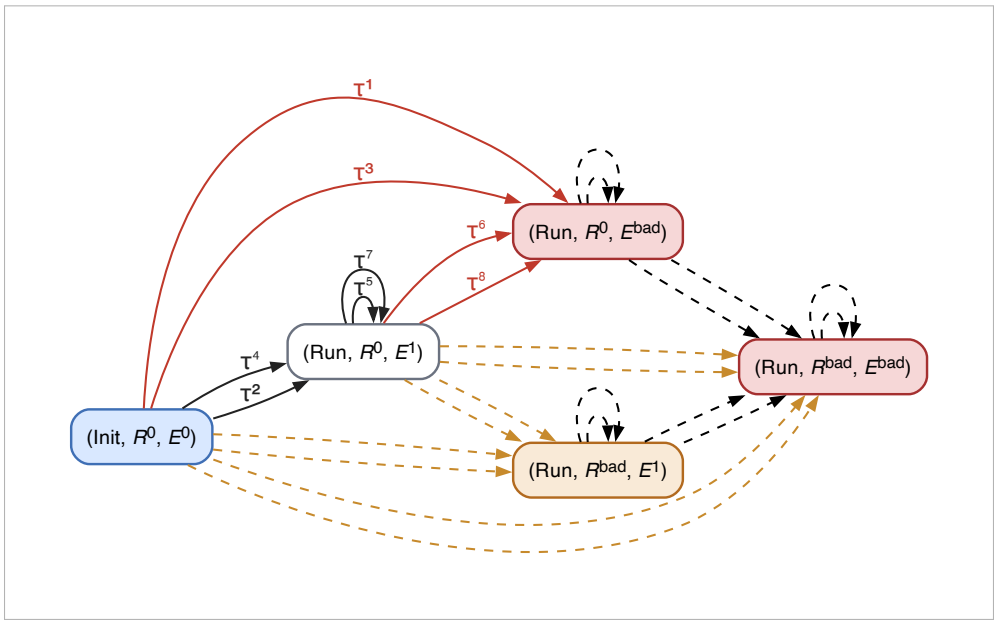
L'IR organise la réduction en obligations locales indépendantes du backend. Étant donné un état de l'automate produit, plusieurs transitions sortantes, sûres ou non sûres, peuvent correspondre à la même transition programme



(a) Automate d'hypothèses



(b) Automate de garanties



(c) Automate produit

FIGURE 2 – Automates du fil rouge.

Automate require	
ϕ^1	$\neg reset = 0 \wedge \neg reset = 1 \vee reset = 1 \wedge \neg x = 0$
ϕ^2	$reset = 1 \wedge x = 0 \vee reset = 0$
ϕ^3	$\top$

Automate ensures	
ψ^1	$\neg reset = 1 \vee y = 0$
ψ^2	$reset = 1 \wedge \neg y = 0$
ψ^3	$reset = 1 \wedge \neg y = 0 \vee reset = 0 \wedge \neg y = 0 \wedge \mathsf{pre}(reset) = 1 \vee reset = 0 \wedge \neg y = \mathsf{pre}(x) \wedge \mathsf{pre}(reset) = 0$
ψ^4	$y = 0 \wedge \mathsf{pre}(x) = 0 \vee \neg reset = 0 \wedge \neg reset = 1 \vee \neg reset = 0 \wedge y = 0$ $\vee y = 0 \wedge \neg \mathsf{pre}(reset) = 0 \vee \neg reset = 1 \wedge \neg \mathsf{pre}(reset) = 0 \wedge \neg \mathsf{pre}(reset) = 1$ $\vee \neg reset = 1 \wedge y = \mathsf{pre}(x) \wedge \neg \mathsf{pre}(reset) = 1$
ψ^5	$\top$

Produit			
τ^1	$(reset = 0, \phi^2, \psi^2)$	τ^5	$(reset = 0, \phi^2, \psi^4)$
τ^2	$(reset = 0, \phi^2, \psi^1)$	τ^6	$(reset = 0, \phi^2, \psi^3)$
τ^3	$(reset = 1, \phi^2, \psi^2)$	τ^7	$(reset = 1, \phi^2, \psi^4)$
τ^4	$(reset = 1, \phi^2, \psi^1)$	τ^8	$(reset = 1, \phi^2, \psi^3)$

TABLE 1 – Correspondance des étiquettes ϕ^i, ψ^i, τ^i.

et à la même composante hypothèse. Elles ne diffèrent que par leur composante garantie. Pour les cas sûrs, cela signifie que, pour un même pas de programme et un même contexte d'entrée, la satisfaction de la spécification admet plusieurs progressions du programme. Ces transitions sont regroupées dans un même *résumé local*. Celui-ci agrège un contexte d'entrée H et une postcondition commune D, tandis que les informations des transitions individuelles sont conservées de façon sélective : pour chaque branche sûre, l'information utile à la transmission temporelle ; pour chaque branche non sûre, l'information utilisée pour les obligations d'exclusion.

Résumé local. Un résumé local S est un n-uplet (ctx, H, D, cas) où :

- $ctx = (P, R, E, t, \phi)$ est le contexte source : P est l'état de contrôle du programme, R l'état courant de l'automate

d'hypothèses, E l'état courant de l'automate de garanties, t la transition programme considérée et ϕ la composante hypothèse ;

- H est le *contexte d'entrée* : il décrit les hypothèses valables en début de pas ;
- D est la *postcondition de sûreté commune* : elle capture la partie commune des branches admissibles ;
- *cas* est la liste des *cas résiduels*. Chaque cas correspond à une transition produit associée à t et ϕ et porte sa classe (sûre ou non sûre), une destination produit, une formule de propagation (`propagages`) pour les transitions sûres et, pour les transitions non sûres, une formule d'exclusion (`forbidden`).

La correction s'appuie sur deux familles d'obligations locales pour chaque résumé. La première est un triplet $\{H\}\ c\ \{D\}$, où c est le fragment de code de la transition du programme. La seconde est, pour chaque cas non sûr, l'obligation d'exclusion $H \wedge \psi^{bad} \Rightarrow \bot$ où ψ^{bad} est la formule d'exclusion (`forbidden`). Les résumés sont construits en trois passes et s'appuient sur des fonctions de décalage temporel. Pour notre exemple, quatre résumés sont générés :

- $ctx = (Init, R^0, E^0)$, transition `Init -> Run` sous $reset = 0$ et ϕ^2, correspondant à τ^1, τ^2 ;
- $ctx = (Run, R^0, E^0)$, transition `Init -> Run` sous $reset = 1$ et ϕ^2, correspondant à τ^3, τ^4 ;
- $ctx = (Run, R^0, E^1)$, transition `Run -> Run` sous $reset = 0$ et ϕ^2, correspondant à τ^5, τ^6 ;
- et $ctx = (Run, R^0, E^1)$, transition `Run -> Run` sous $reset = 1$ et ϕ^2, correspondant à τ^7, τ^8.

Décalages temporels. La construction des résumés repose sur des opérations de décalage temporel des formules : shift qui transporte vers le début du pas courant des informations calculées à la fin du pas précédent et shift^{-1} qui est l'opération inverse. Ces décalages portent sur (1) les variables d'entrée et (2) les variables de sortie/locales déjà décalées dans les formules. La distinction vient du fait que les valeurs des sorties/locales sont préservées entre la fin d'un pas et le début du suivant. L'opération inverse n'est définie que pour des valeurs décalées d'au moins un instant, ce qui n'est pas un problème car elle n'est utilisée que sur les invariants d'états pour lesquels Kairos interdit de mentionner l'entrée courante.

Passe 1 : initialisation. Cette phase initialise les composantes `propagages` (cas sûrs) et `forbidden` (cas non sûrs) avec la composante garantie de la transition produit considérée. Pour notre exemple, en considérant le cas (Run, R^0, E^1) pour la transition programme `Run -> Run` sous $reset = 0$ et la composante hypothèse ϕ^2, on obtient

- un unique cas sûr (dst = (Run, R^0, E^1), `propagages` = ψ_4) correspondant à τ^5,
- et un unique cas non sûr (dst = (Run, R^0, E^{bad}), `forbidden` = ψ_3) correspondant à τ^6.

Cette initialisation est globale car le calcul ultérieur de H dépend des propagations issues des résumés des transitions entrantes. Les composantes H et D ne sont pas initialisées à ce stade.

Passe 2 : Contexte d'entrée. Cette passe construit le contexte d'entrée local H. Elle combine d'abord deux contraintes structurelles du pas courant : la composante hypothèse et la garde de transition du programme (ici, ϕ^2 et $(reset = 0)$ respectivement). À ces contraintes locales, s'ajoutent les contraintes de stabilité des sorties et variables locales, ainsi que l'invariant de l'état courant. L'héritage temporel est obtenu par disjonction des informations de propagation des transitions sûres entrantes, décalées par `shift` (d'où la nécessité de réaliser globalement la phase initialisation). Pour notre fil rouge, on obtient :

$$
\begin{aligned}
H(S) &= \chi_1 \wedge \chi_2 \wedge \chi_3 \wedge \chi_4, \\
&\chi_1 = \phi_2 \wedge (reset = 0), \quad \chi_2 = (m = \mathsf{prev}\ 1\ m) \wedge (y = \mathsf{prev}\ 1\ y), \\
&\chi_3 = \mathsf{shift}(\psi_1) \vee \mathsf{shift}(\psi_4), \quad \chi_4 = m = \mathsf{prev}\ 1\ x.
\end{aligned}
$$

La composante χ_1 exprime la composante hypothèse et la garde de transition du programme. La composante χ_2 exprime la stabilité des sorties/locales entre fin et début de pas. La composante χ_3 provient des branches sûres entrantes en (Run, R^0, E^1) : les transitions τ^2 et τ^4 apportent la formule ψ_1, tandis que τ^5 et τ^7 apportent ψ_4 ; la passe shift aligne ensuite cette information au début du pas courant. La composante χ_4 correspond à l'invariant du nœud courant. La forme développée de la composante temporelle est reportée dans la Figure 3.

Passe 3 : Postcondition de sûreté. Cette passe construit la composante commune D des transitions sûres sortantes du résumé, obtenue par disjonction des gardes de garantie sûres, à laquelle s'ajoute l'invariant de la destination de t (ramené en fin du pas courant par shift^{-1}). Sur ce fil rouge, aucune disjonction multi-branche n'apparaît car l'automate de garantie n'a pas de nœud avec plusieurs transitions sûres sortantes. Pour le fil rouge, l'invariant destination décalé vaut $m = x$, et l'on obtient simplement :

$$D(S) = \psi_4 \wedge (m = x).$$

$$\mathsf{shift}(\psi_1) = \neg(\mathsf{prev}\ 1\ reset = 1) \vee y = 0.$$

$$\begin{aligned}\mathsf{shift}(\psi_4) = {} & y = 0 \wedge \mathsf{prev}\ 2\ x = 0 \\ & \vee \neg(\mathsf{prev}\ 1\ reset = 0) \wedge \neg(\mathsf{prev}\ 1\ reset = 1) \\ & \vee \neg(\mathsf{prev}\ 1\ reset = 0 \wedge y = 0) \\ & \vee y = 0 \wedge \neg(\mathsf{prev}\ 2\ reset = 0) \\ & \vee \neg(\mathsf{prev}\ 1\ reset = 1) \wedge \neg(\mathsf{prev}\ 2\ reset = 0) \wedge \neg(\mathsf{prev}\ 2\ reset = 1) \\ & \vee \neg(\mathsf{prev}\ 1\ reset = 1) \wedge y = \mathsf{prev}\ 2\ x \wedge \neg(\mathsf{prev}\ 2\ reset = 1).\end{aligned}$$

Figure 3 – Formes développées des décalages $\mathsf{shift}(\psi_1)$ et $\mathsf{shift}(\psi_4)$.

Synthèse. Après les trois passes, le résumé local S est entièrement déterminé, ainsi que le triplet et la condition d'exclusion associés. Kairos ajoute en outre, lorsque l'état initial porte un invariant, un goal dédié pour établir le cas de base au premier pas, où aucun héritage temporel par décalage ne peut encore être mobilisé ; ce goal est absent sur ce fil rouge, qui ne spécifie pas d'invariant sur l'état initial.

$$\{H(S)\}\ \texttt{y:=m; m:=x}\ \{D(S)\} \qquad H(S) \wedge \psi_3 \Rightarrow \bot$$

Finalement, Kairos produit, à partir des résumés locaux, un programme Why3 où les expressions de la forme `prev k x` sont transformées en variables logiques de la forme `prev_k_x`. L'API Why3 est finalement utilisée pour générer des obligations de vérification, déchargées par le solveur SMT Z3 [13]. Le fil rouge présenté ici est validé par cette chaîne.

5 Mesures

Nous présentons une micro-campagne expérimentale sur six exemples validés afin de documenter le comportement de la chaîne de réduction. L'objectif n'est pas de fournir un benchmark exhaustif, mais des ordres de grandeur sur un prototype mono-thread de Kairos. Le `README` du dépôt indique les commandes de reproduction (génération des artefacts, exécution des campagnes, reconstruction des tableaux). Les six exemples mesurés sont issus du corpus public `tests/ok` ; le dépôt fournit également un corpus de contre-exemples `tests/ko`. Les métriques du tableau principal sont : $|S_p|$ et $|T_p|$, nombres de nœuds et transitions du produit complet ; $|S_u|$ et $|T_u|$, nombres de nœuds et transitions utiles (sous-graphe vivant effectivement exploité par l'IR) ; $|S|$, nombre de résumés locaux ; $|\kappa_{safe}|$, nombre total de transitions sûres ; $|\kappa_{bad}|$, nombre total de transitions non sûres ; et $|Obl|$, nombre total de VCs (conditions de vérification) générées (triplets + conditions d'exclusion, mesurés sur le dump VC Why3).

Hors fil rouge, les cinq exemples additionnels couvrent : `edge_rise` (détection de front montant), `toggle` (alternance stricte 0/1 de la sortie), `handoff` (bascule entre deux modes exclusifs), `armed_delay_flag` (armement conditionnel puis maintien d'un délai d'un pas), et `w_guarded_prev_hold` (contrainte historique sous `Weak-Until`). La Table 2 montre, sur cet échantillon, que le produit complet peut être sensiblement plus grand que sa partie utile

Exemple	Produit complet		Produit utile		Résumés locaux	Cas		Obligations
	$\lvert S_p\rvert$	$\lvert T_p\rvert$	$\lvert S_u\rvert$	$\lvert T_u\rvert$	$\lvert S\rvert$	$\lvert\kappa_{safe}\rvert$	$\lvert\kappa_{bad}\rvert$	$\lvert Obl\rvert$
`resettable_delay`	5	26	2	8	4	4	4	12
`armed_delay_flag`	7	24	5	21	8	13	8	24
`edge_rise`	3	5	2	4	2	2	2	9
`handoff`	11	30	9	28	9	22	6	24
`toggle`	5	12	4	11	4	7	4	8
`w_guarded_prev_hold`	13	88	5	26	10	18	8	23

Table 2 – Résumé quantitatif

(par exemple `w_guarded_prev_hold` : $|S_p| = 13, |T_p| = 88$ contre $|S_u| = 5, |T_u| = 26$), tandis que la charge de preuve reste contenue ($|Obl| = 23$). Sur le fil rouge, on obtient $|S_p| = 5, |T_p| = 26, |S_u| = 2, |T_u| = 8, |S| = 4, |Obl| = 12$. La différence entre $|S|$ et $|Obl|$ est attendue : chaque résumé local engendre un triplet, les cas non sûrs ajoutent des goals d'exclusion, et une VC peut encore être décomposée en plusieurs buts (postconditions conjonctives). Ainsi, $|Obl|$ mesure la charge effective de preuve, et non un simple comptage des résumés locaux. En complément, la Table 3 isole cinq postes de coût : Spot, produit, IR, génération Why3 et VC+SMT. La colonne *Total* sert de repère de bout en bout, sans prétendre épuiser tout l'overhead frontend/orchestration. Sur cet échantillon, VC+SMT domine nettement, les autres postes restant plus modestes. Les expérimentations ont été réalisées sur une machine Apple M5 (15 cœurs CPU, 24 Go RAM). Kairos étant un premier prototype, les phases produit et IR offrent encore plusieurs pistes d'optimisation (structures de données, exploration, factorisation).

Exemple	Temps par phase (s)					Total (s)
	Spot	Produit	IR	Génération Why3	VC+SMT	
`resettable_delay`	0.042	0.000038	0.000012	0.002	0.706	0.736
`armed_delay_flag`	0.025	0.000039	0.000019	0.002	0.909	0.926
`edge_rise`	0.023	0.000014	0.000007	0.000	0.644	0.657
`handoff`	0.023	0.000036	0.000019	0.002	0.859	0.875
`toggle`	0.022	0.000021	0.000007	0.000	0.624	0.636
`w_guarded_prev_hold`	0.044	0.000090	0.000017	0.002	0.955	0.988

Table 3 – Complément temporel

6 Portée et limites

La réduction présentée cible les spécifications LTL conditionnelles de sûreté dont les atomes sont des formules du premier ordre sur l'historique borné. Dans ce cadre, la méthode fournit une chaîne explicite entre clauses globales, contextes produit, résumés locaux et obligations locales. Elle est particulièrement adaptée aux propriétés historiques conditionnelles, où le diagnostic exige de relier précisément une violation locale à son origine temporelle dans la trace. Trois limites bornent le domaine d'efficacité actuel. Premièrement, le fragment de sûreté traité exclut les propriétés de vivacité générales : un opérateur $F(\phi)$ non borné ne peut pas être encodé dans les automates de sûreté utilisés. Deuxièmement, la scalabilité du produit dépend du nombre d'états des automates $\mathcal{A}$ et $\mathcal{G}$: des spécifications complexes peuvent produire un produit de grande taille, même si la condensation en résumés locaux en réduit significativement l'impact sur les obligations finales (cf. ratios $|T_p|/|Obl|$ de la Table 2). Une étude expérimentale plus approfondie, reposant sur une extension du langage, sera nécessaire pour mieux caractériser cette problématique, étroitement liée aux questions de modularité (voir conclusion). Troisièmement, la qualité du diagnostic dépend étroitement de la richesse des invariants d'état fournis par l'utilisateur : des invariants trop faibles conduisent à des obligations de preuve qui ne peuvent pas être déchargées automatiquement, sans que la réduction elle-même soit en cause. Ce phénomène est analogue à celui rencontré lors de la spécification d'invariants de boucle.

7 Travaux connexes

La traduction LTL/automate de sûreté utilisée par Kairos s'appuie sur des fondements établis [3, 7, 8, 5] et sur l'outil Spot [12] pour la synthèse effective. La différence avec le model checking classique [8] est la cible de la traduction : non pas la décision directe sur l'ensemble des traces, mais la construction d'obligations locales de pas, destinées à un backend de preuve déductive. Cette orientation locale requiert en particulier le calcul explicite des contextes d'entrée hérités, qui est absent des approches purement fondées sur les automates. L'approche de Kairos prolonge la tradition Lustre/Hoare [1, 2, 14, 15, 16] : spécifier les programmes réactifs par des contrats temporels, puis les vérifier par preuve locale. La spécificité est la chaîne de traçabilité rendue explicite, du produit synchronisé au résumé local jusqu'à la VC finale, avec les décalages historiques shift calculés automatiquement. Dans l'approche Lustre/PVS [14], ces connexions sont établies manuellement ; Kairos les automatise via le middle-end IR. Kind 2 [17] et AGREE [18] réalisent une vérification compositionnelle par k-induction ou model checking symbolique, avec un haut degré d'expressivité et d'automatisation. L'usage visé par Kairos est complémentaire : rendre l'obligation locale interprétable et la trace de diagnostic directement reliée à la clause source, dans des situations où l'utilisateur doit comprendre précisément pourquoi une obligation échoue et quelles hypothèses historiques sont manquantes. Par ailleurs, l'approche déductive n'est pas soumise aux contraintes d'un domaine abstrait fixé : le noyau de langage peut manipuler des types arbitraires — tableaux, structures, arithmétique réelle — dès lors que le backend de preuve, ici Why3 [9], sait les traiter. Cette extensibilité par le backend est structurellement absente des approches par model checking symbolique, qui restent liées à la décidabilité du domaine considéré.

8 Conclusion

Cet article a présenté, au travers d'un fil rouge, l'outil Kairos comme une chaîne de réduction sémantique transformant des contrats LTL conditionnels en obligations locales de pas. Le principe est le suivant : la validité conjointe des obligations de sûreté et d'exclusion entraîne la satisfaction de la spécification temporelle d'origine. Cette réduction combine la construction d'automates de sûreté, la formation d'un automate produit et la contextualisation par les opérateurs de décalage shift et shift^{-1}, qui rendent localement exploitables les dépendances historiques.

L'outil Kairos permet ainsi d'établir, par des techniques de vérification déductive, la correction de programmes réactifs vis-à-vis de spécifications temporelles exprimées en LTL. Par rapport aux approches fondées sur le model checking, Kairos autorise l'extension de la vérification à ce type de programmes dans des contextes où l'expressivité et les domaines de données ne sont limités que par le pouvoir expressif du backend déductif utilisé. Cette approche ouvre également la voie au développement de techniques de preuve modulaires. La contribution de Kairos est également architecturale : la sémantique de la réduction est portée par l'IR du middle-end, tandis que le backend intervient uniquement comme moteur terminal de décharge des obligations. Une version abstraite de ce principe de

réduction a été formalisée et prouvée dans Rocq, pour un langage ne comportant pas encore d'automates de modes.

Parmi les perspectives de travaux futurs, nous visons une formalisation dans Rocq plus proche du langage Kairos et de la chaîne effectivement implémentée. Nous souhaitons également étendre le fragment impératif afin d'accroître l'expressivité du langage. Deux sources d'inspiration possibles sont le langage de programmation des plateformes Arduino et le langage WhyML du système Why3. Nous envisageons également d'enrichir les opérateurs portant sur l'historique des exécutions. Une première étape consistera à étudier les fonctions sur l'historique dont la définition inductive permet une réduction à des invariants locaux. Enfin, un enjeu majeur sera l'étude de la preuve modulaire, en étendant le langage afin qu'un nœud puisse inclure des instances d'autres nœuds, comme cela est le cas dans le langage OBC. Dans ce cadre, un nœud fait progresser ses instances en invoquant leur fonction de calcul de pas à chaque étape d'exécution. Nous espérons exploiter les résumés calculés pour chaque nœud afin de réaliser cette composition de manière modulaire.

Références

[1] Nicolas Halbwachs. *Synchronous Programming of Reactive Systems*. Springer, 1993. doi : 10.1007/978-1-4757-2231-4.

[2] Albert Benveniste et Gérard Berry. "The Synchronous Approach to Reactive and Real-Time Systems". In : *Proceedings of the IEEE* 79.9 (1991). doi : 10.1109/5.97297.

[3] Amir Pnueli. "The Temporal Logic of Programs". In : *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*. FOCS '77. IEEE Computer Society, 1977. doi : 10.1109/SFCS.1977.32.

[4] Edmund M. Clarke, Orna Grumberg et Doron A. Peled. *Model Checking*. MIT Press, 1999. isbn : 9780262038836.

[5] Paul Gastin et Denis Oddoux. "Fast LTL to Büchi Automata Translation". In : *Computer Aided Verification*. T. 2102. Lecture Notes in Computer Science. Springer, 2001. doi : 10.1007/3-540-44585-4_6.

[6] Moshe Y. Vardi et Pierre Wolper. "An Automata-Theoretic Approach to Automatic Program Verification". In : *Proceedings of the First Symposium on Logic in Computer Science*. 1986, p. 332-344.

[7] Bowen Alpern et Fred B. Schneider. "Defining Liveness". In : *Information Processing Letters* 21.4 (1985). doi : 10.1016/0020-0190(85)90056-0.

[8] Orna Kupferman et Moshe Y. Vardi. "Model Checking of Safety Properties". In : *Formal Methods in System Design* 19.3 (2001). doi : 10.1023/A:1011254632723.

[9] Jean-Christophe Filliâtre et Andrei Paskevich. "Why3 — Where Programs Meet Provers". In : *Programming Languages and Systems*. T. 7792. Lecture Notes in Computer Science. Springer, 2013. doi : 10.1007/978-3-642-37036-6_8.

[10] Timothy Bourke et al. "A Formally Verified Compiler for Lustre". In : *Conference on Programming Language Design and Implementation*. PLDI. ACM, 2017. doi : 10.1145/3062341.3062358.

[11] Florence Maraninchi et Yann Rémond. "Mode-Automata : a new domain-specific construct for the development of safe critical systems". In : *Science of Computer Programming* 46.3 (2003). Special issue on Formal Methods for Industrial Critical Systems. doi : https://doi.org/10.1016/S0167-6423(02)00093-X.

[12] Alexandre Duret-Lutz et al. "From Spot 2.0 to Spot 2.10 : What's New ?" In : *Computer Aided Verification*. T. 13372. Lecture Notes in Computer Science. Springer, 2022. doi : 10.1007/978-3-031-13188-2_9.

[13] Leonardo de Moura et Nikolaj Bjørner. "Z3 : An Efficient SMT Solver". In : *Tools and Algorithms for the Construction and Analysis of Systems*. T. 4963. Lecture Notes in Computer Science. Springer, 2008. doi : 10.1007/978-3-540-78800-3_24.

[14] Saddek Bensalem et al. "A Methodology for Proving Control Systems with Lustre and PVS". In : *Dependable Computing for Critical Applications—7*. T. 12. Dependable Computing and Fault Tolerant Systems. IEEE Computer Society, 1999. doi : 10.1109/DCFTS.1999.814291.

[15] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". In : *Communications of the ACM* 12.10 (1969). doi : 10.1145/363235.363259.

[16] Stephen A. Cook. "Soundness and Completeness of an Axiom System for Program Verification". In : *SIAM Journal on Computing* 7.1 (1978). doi : 10.1137/0207005.

[17] Adrien Champion et al. "The Kind 2 Model Checker". In : *Computer Aided Verification*. T. 9780. Lecture Notes in Computer Science. Springer, 2016. doi : 10.1007/978-3-319-41540-6_29.

[18] John Backes et al. *The Assume Guarantee Reasoning Environment*. Rapp. tech. Loonwerks, 2021. url : https://loonwerks.com/publications/pdf/backes2021techreport.pdf.