Vers une conception de systèmes réactifs synchrones sûrs

Sarah Chabane[1], Rabea Ameur-Boulifa[2], and Mezghiche Mohamed[1]

[1] Laboratoire LIMOSE, département d'informatique
Faculté des sciences
Université M'hamed Bougara, Boumerdes, Algérie
chabane.sarah@univ-boumerdes.dz
mohamed.mezghiche@gmail.com
[2] LTCI, Télécom Paris, Institut polytechnique de Paris
rabea.ameur-boulifa@telecom-paris.fr

Résumé

La nécessité de gérer la complexité croissante des systèmes en général, et les systèmes embarqués en particulier pousse leur conception vers une approche basée sur la réutilisation de composants existants. À cette complexité s'ajoute des exigences techniques, les systèmes doivent satisfaire des contraintes strictes de fiabilité et de correction. Dans notre travail, nous préconisons d'offrir un cadre théorique pour le développement de composants réactifs synchrones sûrs de manière compositionnelle. Dans ce papier nous offrons d'une part, un cadre de description de composants réactifs synchrones élémentaires dans un formalisme adapté pour la vérification formelle de propriétés de sûreté, et d'autre part, un cadre pour la construction des systèmes globaux à partir de composants élémentaires par une opération de composition garantissant la correction par construction.

1 Introduction

La complexité croissante des circuits et des systèmes embarqués représente un enjeu majeur pour les concepteurs de ces systèmes. La réutilisabilité de composants existants s'avère une approche efficace et largement utilisée dans divers projets pour contrer cette complexité. D'autre part, l'utilisation des méthodes formelles dans le cadre du développement de systèmes apporte une rentabilité plus élevée, par la réduction des coûts et des délais de validation, et aussi une fiabilité plus élevée, par l'efficacité de ces méthodes. En effet, l'approche formelle s'appuie sur des formalismes et des techniques permettant de raisonner rigoureusement sur des systèmes. Combiner les deux approches apparaît naturellement un bon moyen pour maitriser la conception des systèmes et garantir leur qualité. La démarche de conception à base de composants est pour sa part corroborée dans le domaine du développement logiciel depuis quelques décennies, elle se focalise plutôt sur la réutilisation du code en adoptant une forte standardisation des composants constitants le système. Dans le contexte de ce travail, le problème est de garantir qu'un système construit à partir d'un ensemble donné de sous-systèmes et d'un ensemble d'opérateurs de composition satisfait une propriété donnée [20]. Dans ce papier nous introduisons un cadre théorique pour la conception de systèmes réactifs synchrones corrects à partir de sous-systèmes. La conception de systèmes dans le cadre proposé s'inscrit dans une approche modulaire compositionnelle en prônant l'utilisation des méthodes formelles pour garantir la conception de systèmes corrects par construction.

Parmi les modèles adaptés et utilisés avec succès pour la modélisation des systèmes basés sur la notion d'évènements tels que les systèmes réactifs, nous trouvons les automates

d'entrées/sorties, dits *I/O-automata* de l'anglais et que nous noterons le long du papier *I/O-automates*, introduits par Lynch dans [16]. La sémantique de ces formalismes focalisée sur les évènements d'entrée et les événements de sortie correspond parfaitement à la sémantique des composants réactifs, elle est bien adaptée pour la description de leurs comportement. De plus, la sémantique des I/O-automates repose intrinsèquement sur le principe de compositionnalité, i.e, des automates globaux peuvent être construits à partir d'automates élémentaires grâce à un opérateur de composition. Toutefois, l'opérateur de composition des I/O-automates s'avère inapproprié pour la modélisation des composants dont le fonctionnement est basé sur le principe du premier entré, premier sorti, nodans ls suite us utiliserons le terme pipeline pour designer ce type de composant. La composition en série de deux pipelines, telle que les sorties du premier permettent d'alimenter directement le second, produit naturellement un pipeline de taille plus grande. Cependant, du point de vue des modèles comme cela a été illustré sur une étude de cas dans [17], la composition des modèles I/O-automates associés à deux pipelines (assemblés en série) ne produit pas le modèle des I/O-automates associé au pipeline résultant, autrement dit, les règles de composition des I/O-automates ne préservent pas la sémantique de composants de type pipeline. Dans ce papier, nous proposons de réviser les règles de composition des I/O-automates afin de garantir la préservation de la sémantique des modèles de composants par l'opération de composition. La contribution de ce papier est double :

- Proposer un cadre pour générer automatiquement des modèles formels pour la spécification des composants réactifs à partir de ses caractéristiques : la taille de la mémoire et le temps de latence. Ces modèles qui s'appuient sur le formalisme des I/O-automates peuvent être utilisés comme base pour prouver automatiquement des propriétés sur les systèmes étudiés, en utilisant les outils de vérification du type model-checking.
- Proposer un opérateur de composition qui préserve par composition la sémantique de composants réactifs, i.e, un opérateur de composition qui permet de composer deux modèles associés à deux pipelines et d'en générer un modèle global correct. Cet opérateur de composition permet de construire de manière compositionnelle et hiérarchique des systèmes corrects par construction à partir de sous-systèmes élémentaires.

La suite de l'article est organisée comme suit : nous rappelons dans la Section 2 la notion de systèmes réactifs synchrones. Dans la Section 3 nous présenterons les modèles des I/O-automates qui sont des modèles sémantiques appropriés pour représenter les systèmes réactifs synchrones; nous expliquerons comment ces modèles sont construits à partir de caractéristiques choisies des systèmes. Nous rappellerons la définition de la composition des I/O-automates, et nous montrerons que cette opération n'est pas adaptée pour la modélisation de la composition des systèmes réactifs synchrones. Nous proposerons alors une révision de cette opération de composition pour garantir la correction. Nous donnons ensuite dans la Section 4 des arguments permettant de valider notre approche. Dans la Section 5 nous présenterons les travaux connexes et conclurons par la Section 6.

2 Systèmes réactifs synchrones

Un système réactif, appelé aussi système à base d'évènements, est un système qui interagit avec son environnement sur la base d'une séquence de réactions. Chaque réaction est déclenchée quand un évènement survient, et a un impact sur l'état du système. Les principes de l'approche réactive peuvent être utilisés comme une approche méthodologique pour la conception et le développement de systèmes concurrents, où les unités de conception sont des composants fonctionnels distribués qui interagissent entre eux grâce à une communication à base d'évènements.

À l'instar des modèles de calcul à déclenchement temporel dit time-triggered Model of

Computation [14], les composants synchrones sont rythmés par une horloge globale qui coordonne l'ensemble des calculs; les calculs prennent du temps et ne sont pas instantanés. Le comportement du composant n'est pas conduit par les évènements qui se produisent dans son environnement, mais décide de manière autonome quand interagir avec l'extérieur.

Dans ce papier, nous nous intéressons aux composants réactifs synchrones (que nous noterons dans la suite SR-composants), ces composants sont régis par une horloge logique sur laquelle repose le mécanisme d'échanges instantanés et simultanés d'information (ou données). Le comportement d'un SR-composant peut être entièrement spécifié par l'ensemble d'états et de séquences permises d'événements d'entrée et de sortie qu'échange le composant avec son environnement. En plus des propriétés générales que doivent satisfaire les SR-composants, comme le déterminisme et la réactivité, ils doivent aussi satisfaire des propriétés spécifiques liées aux caractéristiques du composant. Dans ce papier nous nous intéressons à deux caractéristiques particulières des SR-composants qui sont la capacité de stockage de données (*mémoire*) et la durée de traitement d'une donnée, i.e, la durée entre une action et le déclenchement d'une réaction (*latence*). Nous considérons qu'un SR-composant est défini par ces deux paramètres que nous noterons resp. $\mathtt{M}$ et $\mathtt{L}$. Nous considérons aussi qu'à chaque cycle d'horloge un composant ne peut consommer qu'une seule donnée en entrée et peut produire une seule donnée en sortie. Une entrée est stockée dans le composant jusqu'à la délivrance de la sortie correspondante. Le comportement de SR-composants ainsi défini peut être décrit par une séquence d'états où chaque état peut être spécifié par une liste de données consommées par le composant et la durée d'attente de chaque donnée avant que la sortie correspondante soit disponible. Concrètement, cette liste représente explicitement le flot de données consommées par le composant à un instant donné, et dont chaque élément de la liste représente non pas la valeur de la donnée mais par une date (*timestamp*) – représentant la quantité de temps écoulée depuis l'entrée de la donnée jusqu'à son effacement, i.e., la délivrance de la sortie correspondante.

Étant donné que les paramètres $\mathtt{L}$ et $\mathtt{M}$ sont des constantes, l'ensemble des états d'un SR-composant est fini. Chaque état s peut être codé par une liste d'entiers comme suit $s = \langle i \rangle^{i \in [0..\mathtt{L}]}$ tel que $\mathit{length}(s) \leq \mathtt{M}$ (avec $\mathit{length}(s)$ la fonction qui retourne la taille de la liste). L'état initial est représenté par la liste vide ($\langle\rangle$). Notons que dans ce codage nous considérons que toutes les données dont la date est supérieure ou égale à la latence sont codées $\mathtt{L}$. En effet, nous montrons que le comportement d'un état ayant des données dont la date dépasse la latence est équivalent à état dont la date de ces données a atteint la latence (voir Section 3.1). Par ailleurs, en considérant que le composant ne consomme et ne produit qu'une seule donnée à chaque tick d'horloge, on en déduit que la séquence de dates est ordonnée et ne peut contenir deux dates de la même valeur, excepté pour les données dont la date est supérieure ou égale la latence. Le nombre d'états d'un SR-composant défini par les paramètres $\mathtt{L}$ et $\mathtt{M}$ peut être calculé par la fonction suivante :

$$\#s(\mathtt{L},\mathtt{M}) = \begin{cases} 1 + \sum_{i=1}^{\mathtt{M}} C_{\mathtt{L}}^{i} & \mathtt{si\ M} \leq \mathtt{L} \\ 1 + \sum_{i=1}^{\mathtt{L}} C_{\mathtt{L}}^{i} + (\mathtt{M} - \mathtt{L}) & \mathtt{autrement} \end{cases}$$

où $C_{\mathtt{L}}^{i}$ désigne la combinaison sans répétition de i dates parmi $\mathtt{L}$. En effet, si $\mathtt{M} \leq \mathtt{L}$ en plus de l'état initial nous pouvons en sélectionner jusqu'à $\mathtt{M}$ dates (ou données). Dans le cas contraire, comme le compteur de date ne dépasse pas la valeur $\mathtt{L}$, nous pouvons en sélectionner jusqu'à $\mathtt{L}$ dates, et toutes les $(\mathtt{M} - \mathtt{L})$ autres dates, elles ont la même valeur $\mathtt{L}$.

3 Modélisation

Pour pouvoir intégrer et utiliser des techniques de vérification formelle dans une approche de conception des systèmes, il est nécessaire d'exprimer ces systèmes dans un formalisme permettant l'analyse formelle. Dans cette section, nous introduisons les modèles choisis pour décrire de manière naturelle le comportement d'un SR-composant. Ces modèles sont utilisés pour vérifier, par model-checking, la satisfaction de propriétés fonctionnelles. Aussi, ces modèles sont adaptés pour une conception de systèmes basée sur une approche compositionnelle, ils peuvent être combinés pour former des modèles complexes cohérents et complets.

3.1 Modèle d'un SR-composant primitif

Pour la description d'un SR-composant nous nous appuyons sur les automates d'entrée/de sortie (I/O-automates) [16]. Ce formalisme présente un grand intérêt pour notre approche, d'une part, il est parfaitement adapté pour représenter la sémantique d'un système réactif, d'autre part, il supporte le principe de compositionnalité : la théorie des automates propose des mécanismes maitrisés pour combiner des automates. Les I/O-automates, comme définis dans la Définition 1, sont des machines à états avec des actions (d'entrée/de sortie) associées aux transitions.

Définition 1 (I/O-automate)**.** *Un I/O-automate est un tuple* $\mathcal{A} \triangleq \langle S, s_0, \Sigma, \rightarrow \rangle$ *où :*

- S *est l'ensemble des états.*
- $s_0 \in S$ *est l'état initial.*
- Σ *est l'ensemble des actions, tel que* $\Sigma = \{i/o, i/\bar{o}, \bar{i}/o, \bar{i}/\bar{o}\}$.
- $\rightarrow \subseteq S \times \Sigma \times S$ *est la relation de transition.*

Chaque transition $(s, l, s') \in \rightarrow$ est notée $s \xrightarrow{l} s'$. Lorsqu'une action l n'est pas exécutable à partit d'un état, on le note par $s \xrightarrow{l} \!\!\!\!/$. L'ensemble $\Sigma = \{i/o, i/\bar{o}, \bar{i}/o, \bar{i}/\bar{o}\}$ représente l'ensemble de toutes les actions que l'automate peut exécuter, il regroupe tous les évènements de présence ou absence, des entrées et des sorties qui peuvent survenir durant l'exécution d'un composant. Informellement, i/o signifie qu'une entrée et une sortie se produisent simultanément; $i/\bar{o}$ signifie qu'une entrée se produit mais qu'aucune sortie n'est produite; $\bar{i}/o$ signifie qu'il n'y a pas d'entrée mais qu'une sortie est produite; et $\bar{i}/\bar{o}$ signifie qu'il n'y a ni entrée, ni sortie.

Bien que le modèle I/O-automate convienne parfaitement pour décrire la sémantique de communication d'un SR-composant avec son environnement. Il ne considère pas les caractéristiques intrinsèques du SR-composant, en particulier sa capacité de stockage et sa réactivité. En effet, en plus des communications le modèle d'un SR composant doit également satisfaire les contraintes suivantes :

- Le composant accepte des données de son environnement tant qu'il y a de l'espace libre, i.e, la capacité de stockage n'est pas épuisée.
- Une entrée doit séjourner dans le composant au moins une période égale à la durée de la latence avant que la sortie correspondante soit produite.
- Une fois qu'une donnée est prête à sortir, le composant a le choix entre la faire sortir ou attendre.

Notations. Avant d'introduire une définition plus appropriée d'un SR-modèle, introduisons quelques notations utiles pour la suite. Nous notons par $\pi = s_0 s_1 s_2 \ldots$ un chemin dans un I/O-automate $\mathcal{A}$ qui est une suite finie ou infinie d'états commençant par l'état initial. La longueur d'un chemin est notée $|\pi| \in \mathbb{N}$, et pour deux états donnés s_i et s_j, nous notons $\pi[s_i] = s_0 \ldots s_i$ une portion du chemin jusqu'à l'état s_i, et $\pi[s_i, s_j]$ la portion du chemin depuis s_i jusqu'à s_j.

Afin de pouvoir compter sur un chemin donné le nombre de transitions correspondant à une action l, nous définissons la fonction filtre qui vérifie si une transition exécute l'action l ou non :

$$\tau(s_i, l, s_{i+1}) = \begin{cases} 1 & \texttt{si } (s_i, l, s_{i+1}) \in \rightarrow \\ 0 & \texttt{autrement} \end{cases}$$

Pour un état donné s on peut calculer le nombre d'entrées stockées dans le composant à cet état, simplement par le comptage sur tout le chemin depuis l'état initial de toutes les entrées (transitions étiquetées $i/\bar{o}$) en soustrayant toutes les sorties (transitions étiquetées $\bar{i}/o$) :

$$m(s) = \sum_{s_i=s_0}^{pre(s)} \tau(s_i, i/\bar{o}, s_{i+1}) - \sum_{s_i=s_0}^{pre(s)} \tau(s_i, \bar{i}/o, s_{i+1})$$

avec $pre(s)$ la fonction qui retourne un état antécédent de s (un état qui précède l'état s).

Notons par s_in_1 la fonction qui retourne l'état où s'effectue la première entrée, i.e, partant de l'état initial le premier état qui exécute une transition i/β. Nous définissons alors la fonction $lfst(s)$ qui calcule à un état donné s (différent de l'état initial) l'âge de la plus vieille donnée par la longueur du chemin le séparant de l'état où a eu lieu la première entrée :

$$lfst(s) = |\pi[s_in_1, s]|$$

En effet, chaque transition représente l'écoulement d'une unité de temps, la durée de séjour d'une donnée peut alors être déterminée par la longueur du chemin parcouru depuis son entrée. La fonction s_in_1 étant non défini à l'état initial s_0 (absence de données à cet état) nous définissons par convention $lfst(s_0) = -1$. Il convient de noter, du point de vue pratique, les états étant codés par des listes, i.e, par le tuple $s = \langle i \rangle^{i \in [1..\mathtt{L}]}$, les fonctions m et $lfst$ correspondent respectivement, à la fonction *length* qui calcule de longueur d'une liste et à la fonction *greatest* qui calcule le plus grand élément d'une liste.

Pour traduire les propriétés comportementales d'un SR-composant sur le modèle I/O-automate, introduisons les prédicats suivants :

- Propriété de *déterminisme* :
  $Determinist(s) \Leftrightarrow \left(\exists s', s'' \in S.(s' \neq s'') \wedge s \xrightarrow{l} s' \wedge s \xrightarrow{l'} s'' \Rightarrow l \neq l'\right)$
  Elle spécifie qu'à partir de chaque état du SR-composant les transitions sont déterminées de façon unique par l'action à exécuter.
- Propriété de *réceptivité* :
  $$Receptiv(s) \Leftrightarrow \begin{cases} (lfst(s) = \mathtt{L} \Rightarrow (\exists s'.s \xrightarrow{\alpha/o} s')) \wedge \\ (m(s) < \mathtt{M} \Rightarrow (\exists s'.s \xrightarrow{i/\beta} s')) \wedge \\ (\exists s' \in S.s \xrightarrow{\bar{i}/\bar{o}} s') \end{cases}$$
  Elle spécifie qu'à chaque état le SR-composant produit des sorties quand elles sont prêtes, accepte des entrées quand elles sont disponibles, et peut aussi décider de rester inactif.
- Propriété de *chronométrage* :
  $OutAfterCount(s) \Leftrightarrow \left(lfst(s) < \mathtt{L} \Rightarrow \nexists s' \in S.s \xrightarrow{\alpha/o} s'\right)$
  Elle spécifie que le SR-composant ne peut produire de sortie d'une entrée n'ayant pas séjourné une période supérieure ou égale à la latence L.
- Propriété de *oisiveté* :
  $IdleAfterCount(s) \Leftrightarrow \left(\exists s' \in S.s \xrightarrow{\bar{i}/\bar{o}} s' \wedge 0 < lfst(s) < \mathtt{L} \Rightarrow s \neq s'\right)$

Elle spécifie que le SR-composant qui a consommé une donnée, et qui n'est donc plus à l'état initial, ne peut rester dans un état oisif (boucler sur lui-même) qu'une fois le décompte de latence terminé, qui correspond à l'arrivée sur un état ayant au moins une donnée prête à sortir.

Nous définissons alors un SR-modèle comme étant un I/O-automate qui satisfait l'ensemble de ces prédicats.

Définition 2 (SR-Modèle)**.** *Considérons C un SR-composant caractérisé par ses paramètres* `L` *et* `M`*, un SR-modèle de C est un I/O-automate $\mathcal{A} = \langle S, s_0, \Sigma, \rightarrow \rangle$ où :*

- $S = \bigcup_{i=0}^{\#s(\mathtt{L},\mathtt{M})} s_i$
- $s_0 = \langle\rangle$*, l'état initial*
- $\Sigma = \{i/o, i/\bar{o}, \bar{i}/o, \bar{i}/\bar{o}\}$
- $\rightarrow$ *est la relation de transition minimale qui satisfait :*
  $\forall s \in S.Determinist(s) \wedge Receptiv(s) \wedge IdleAfterCount(s) \wedge OutAfterCount(s)$

Intuitivement, un SR-modèle est un I/O-automate contraint par le nombre d'états atteignables et l'ensemble de transitions pouvant être exécutées à chaque état. Notons que nous considérons la portée temporelle d'une transition d'un SR-modèle égale à une période d'horloge.

Construction d'un SR-modèle. Cette définition a été implémentée par un algorithme donné en annexe A. L'algorithme prend comme entrée, les paramètres `L` et `M` d'un SR-composant et retourne le SR-modèle correspondant. Son implémentation s'appuie sur le codage proposé, à savoir, les états sont représentés par des listes, et dont chaque élément de la liste i représente la durée de séjour de l'entrée correspondante. L'algorithme simule le comportement d'un SR-composant en démarrant à partir de la liste vide $\langle\rangle$, à chaque nouvelle entrée la liste est complétée d'un élément (de valeur 1) et à chaque sortie la plus ancienne valeur (qui est forcément `L`) est supprimée. Et lorsqu'une nouvelle transition est créée d'un état vers un autre état, tous les éléments de l'état source qui sont reportés dans l'état cible sont incrémentés de 1 – ce qui traduit l'écoulement du temps. Comme nous l'avons mentionné précédemment, par convention, la durée d'attente (de séjour) d'une entrée est bornée par la valeur de la latence `L` ; l'incrémentation est alors appliquée qu'aux éléments de la liste dont la valeur est inférieure à `L`. En effet, nous remarquons que le codage adopté génère des états équivalents : des états qui sont structurellement différents mais du point de vue d'un observateur extérieur, ils simulent un même comportement du SR-composant (ils permettent l'exécution exactement de mêmes transitions). Cela correspond à des états ayant consommé le même nombre d'entrées et pouvant produire la même séquence de sorties, i.e, les sorties sont produites aux mêmes dates. Cette équivalence liée à la date de sortie des données, en l'occurrence à la latence, que nous appelons $\mathcal{L}$-équivalent et que nous notons $s_1 \equiv_l s_2$ (qui se lit deux états s_1 et s_2 sont $\mathcal{L}$-équivalent vis-à-vis d'une date l) est définie comme suit :

$$s_1 \equiv_l s_2 \Leftrightarrow \begin{cases} length(s_1) = length(s_2) \wedge \\ (\forall i \in 1 \ldots length(s_1).(s_1[i] = s_2[i]) \vee (s_1[i] \geq (l - (i-1)) \wedge s_2[i] \geq (l - (i-1))) \end{cases}$$

En utilisant cette notion $\mathcal{L}$-équivalents vis-à-vis de la latence `L` notre algorithme génère le plus petit I/O-automate représentant le comportement d'un SR-composant.

Exemple 1. *Considérons un SR-composant avec les paramètres* $\mathtt{L} = 2$ *et* $\mathtt{M} = 2$.

Le SR-modèle correspondant, généré par l'algorithme (voir Figure 1 droite), est un I/O-automate qui respecte la sémantique du composant. En effet, à partir de l'état initial, représenté par le tuple vide $\langle\rangle$, seules deux transitions sont possibles, la transition $(\bar{i}/\bar{o})$ correspondant à

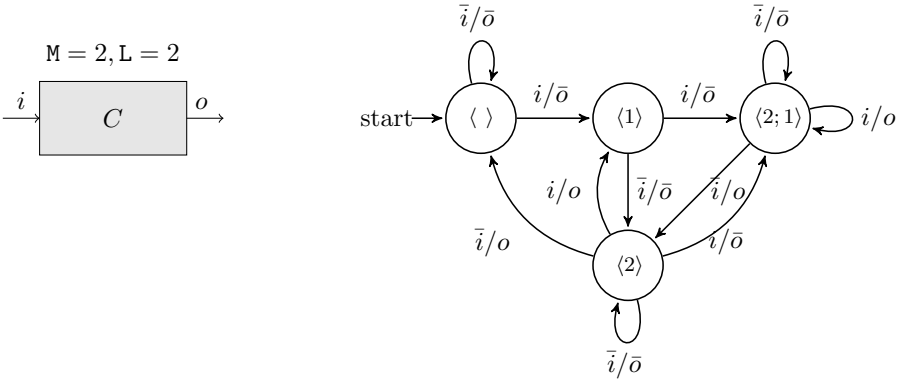

Figure 1: Un SR-composant avec $\mathtt{L} = 2$ et $\mathtt{M} = 2$ et le SR-modèle associé

une attente, et le composant reste sur le même état; ou la transition $(i/\bar{o})$ *correspondant à une consommation d'une entrée et conduisant à un nouvel état* $\langle 1\rangle$*, i.e, l'état où le composant compte une seule entrée consommée depuis une unité de temps. À partir de l'état* $\langle 1\rangle$ *la mémoire n'étant pas pleine, le composant peut donc accepter une autre entrée (transition* $i/\bar{o}$*), et aller vers l'état* $\langle 2; 1\rangle$ *(avec la mémorisation de la date de la nouvelle entrée et incrémentation des anciennes entrées), il peut également continuer à attendre mais pas sur le même état (transition* $\bar{i}/\bar{o}$*) et aller vers l'état* $\langle 2\rangle$ *(comptant ainsi la durée de séjour des entrées consommées). À l'état* $\langle 2\rangle$ *la mémoire n'étant toujours pas pleine, le composant peut accepter une nouvelle entrée sans produire de sortie* $i/\bar{o}$ *et aller vers* $\langle 2; 1\rangle$*, qui est un état dans lequel le composant est plein et à partir duquel celui-ci ne peut plus accepter d'autres entrées sans produire de sortie. À l'état* $\langle 2\rangle$*, le composant dispose d'une entrée qui a déjà atteint la latence, il peut produire une sortie (transition* $\bar{i}/o$ *ou transition* i/o*); il peut continuer d'attendre et rester sur le même état (transition* $\bar{i}/\bar{o}$*) – fin de décompte de la latence; il peut également accepter une nouvelle entrée (transition* $i/\bar{o}$*) et aller vers l'état* $\langle 2; 1\rangle$*. Notons qu'à partir de l'état* $\langle 2; 1\rangle$ *une attente (transition* $\bar{i}/\bar{o}$*) devrait mener vers l'état* $\langle 2; 2\rangle$*, cependant celui-ci étant* $\mathcal{L}$*-équivalent à* $\langle 2; 1\rangle$ *vis-à-vis de* $\mathtt{L} = 2$ *il n'est alors pas généré (la transition boucle sur* $\langle 2; 1\rangle$*).*

3.2 Composition

En général, les systèmes réactifs sont constitués d'un ensemble de composants qui communiquent entre eux. Ils sont construits par assemblage en parallèle ou en série de composants primitifs. Comme illustré par la Figure 2, un SR-composant peut être construit à partir de plusieurs SR-composants assemblés en série. La capacité de stockage du SR-composant résultant est la somme des capacités de stockage de ses sous SR-composants et la latence la somme des latences. De manière générale, l'assemblage en série n SR-composants ayant comme paramètres $\mathtt{L}_i$ et $\mathtt{M}_i$ construit un SR-composant de paramètres $\mathtt{L} = \sum_{i=1}^{n} \mathtt{L}_i$ et $\mathtt{M} = \sum_{i=1}^{n} \mathtt{M}_i$.

L'approche présentée dans ce papier se base sur une méthode d'analyse de SR-composants à travers l'utilisation des automates. Afin de construire le modèle global associé à un SR-composant à partir des modèles associés à ses sous SR-composants, nous définissons une opération de composition. La théorie des I/O-automates est munie d'une opération de composition ; cette opération (rappelée dans la Définition 3) définit la composition par des règles de synchronisation des évènements des automates opérandes : les états deux automates se com-

posent s'il y a un accord sur le statut de leurs entrées et sorties. L'accord se base le principe de communication par handshake (poignée de main) convient naturellement pour la représentation de la composition de SR-composants assemblés en série.

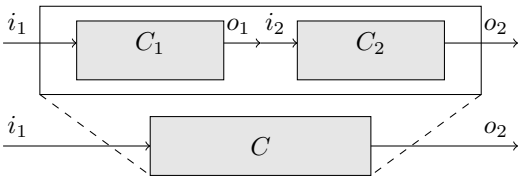

Figure 2: Composition en série de deux composants

Définition 3 (Composition des *I/O-automates*)**.** *Soit deux I/O-automates* $\mathcal{A}_1 = \langle S_1, s_{0_1}, \Sigma, \rightarrow_1 \rangle$ *et* $\mathcal{A}_2 = \langle S_2, s_{0_2}, \Sigma, \rightarrow_2 \rangle$*, leur composition est un I/O-automate* $\langle S_1 \times S_2, (s_{0_1}, s_{0_2}), \Sigma, \rightarrow \rangle$ *tel que* $S_1 \times S_2$ *est l'ensemble des états,* Σ *est l'ensemble des étiquettes,* (s_{0_1}, s_{0_2}) *est l'état initial et* $\rightarrow$ *la fonction de transition définie par les règles suivantes :*

$$\rightarrow \triangleq \begin{cases} (s_1, s_2) \xrightarrow{\alpha/\beta} (s_1', s_2') & \textit{if } (s_1 \xrightarrow{\alpha/o} s_1' \wedge s_2 \xrightarrow{o/\beta} s_2') \vee (s_1 \xrightarrow{\alpha/\bar{o}} s_1' \wedge s_2 \xrightarrow{\bar{i}/\beta} s_2') \\ \textit{indéfini} & \textit{autrement} \end{cases}$$

où α et β représentent resp. les différents statuts des entrées et sorties, i.e, $\alpha = i$ ou $\alpha = \bar{i}$ et $\beta = o$ ou $\beta = \bar{o}$.

En revanche, l'opération de composition des I/O-automates n'est pas adaptée à la composition des SR-modèles. Comme cela a été illustré par un exemple dans [17], le modèle résultant de la composition de modèles associés à des SR-composants assemblés en série (par application des règles de la Définition 3) ne correspond pas au modèle associé au SR-composant global : il ne satisfait pas les propriétés comportementales du SR-modèle attendu. Le problème, décrit de manière détaillée dans [7], se résume comme suit :

- d'une part, la composition génère des modèles non-déterministes. En effet, les deux règles de composition $(s_1 \xrightarrow{\alpha/o} s_1' \wedge s_2 \xrightarrow{i/\beta} s_2')$ et $(s_1 \xrightarrow{\alpha/\bar{o}} s_1'' \wedge s_2 \xrightarrow{\bar{i}/\beta} s_2'')$ peuvent s'appliquer simultanément et entrainer donc la construction de façon concomitante de la transition $(s_1, s_2) \xrightarrow{\alpha/\beta} (s_1', s_2')$ et la transition $(s_1, s_2) \xrightarrow{\alpha/\beta} (s_1'', s_2'')$;
- d'autre part, la composition génère des modèles comportant des états non désirables et manque également de générer des états attendus. Les états non désirables correspondent essentiellement – au cas où le premier composant ne peut plus accepter d'entrées (par manque d'espace de stockage) ni produire de sortie et le second dispose de mémoire, cependant aucune des règles (Définition 3) n'est applicable pour générer l'état composite qui devrait naturellement accepter une entrée ; ou – le cas où le premier composant est prêt à produire une sortie et le second ne peut pas l'accepter (par manque d'espace de stockage), ce qui cause un retard dans le calcul du temps de séjour des entrées du premier composant dans l'état composite.

Concrètement, le problème vient du fait que les SR-composants se comportent comme des buffers de type pipeline, et la composition des I/O-automates ne préserve pas cette sémantique : la composition crée des bulles dans le composant résultant.

Du point de vue de l'implémentation, l'état obtenu par composition de deux états $s_1 = \langle i_1, \ldots, n_1 \rangle$ et $s_2 = \langle i_2, \ldots, n_2 \rangle$ est une liste obtenue par la fusion des listes associées aux états, encodée comme suit : $\langle i_1, \ldots, n_1, i_2 + \mathtt{L}_1, \ldots, n_2 + \mathtt{L}_1 \rangle$. En effet, dans l'état composite toutes

les entrées associées au premier composant conservent leurs valeurs, celles associées au second composant sont additionnées à la latence du premier composant, ajout justifié par la nécessité pour toute entrée atteignant le second composant de traverser d'abord le premier composant.

3.2.1 Composition correcte.

Afin de résoudre le problème de l'inadéquation de l'opération de composition des I/O-automates pour composer SR-modèles. Nous proposons un nouvel opérateur de composition de SR-modèles ; Cet opérateur basé sur le même principe de composition des I/O-automates mais il interdit la génération des états indésirables et contraint la construction des états manquants.

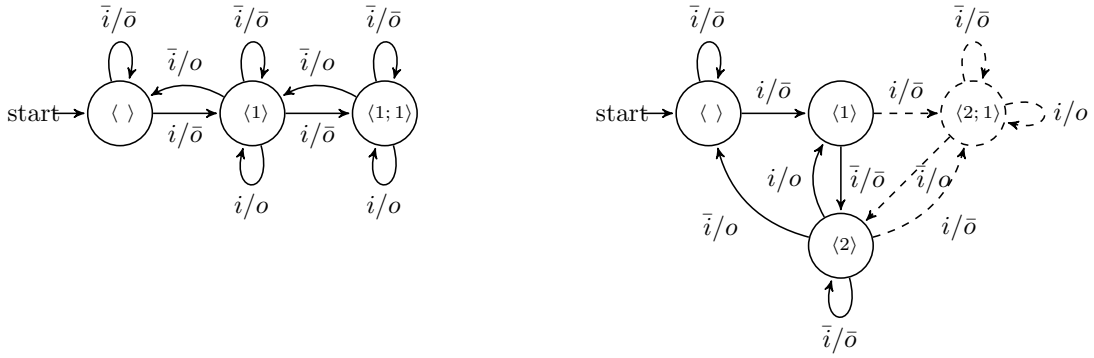

Figure 3: Le SR-modèle correspondant au SR-composant avec $\mathtt{L} = 1$ et $\mathtt{M} = 2$ (à gauche). Un SR-modèle virtuel associé au SR-modèle du SR-composant avec $\mathtt{L} = 2$ et $\mathtt{M} = 1$ (à droite)

La solution proposée repose sur une notion de modèle dit *virtuel*. Un modèle virtuel associé à un SR-composant est le SR-modèle associé au composant, avec une mémoire étendue. Par exemple, l'automate de la Figure 3 (à droite) est un SR-modèle virtuel associé au SR-composant ayant les paramètres $\mathtt{L} = 2$ et $\mathtt{M} = 1$. En effet, l'automate dessiné avec des lignes continues est le SR-modèle associé à ce composant, remarquons que celui-ci est étendu par des états et des transitions (partie dessinée en pointillé) modélisant l'extension de la capacité de stockage d'une unité de mémoire. L'automate complet ainsi obtenu (partie en lignes continues et pointillées) forme le SR-modèle virtuel associé à ce composant. Concrètement, le SR-modèle complet est le SR-modèle associé au SR-composant ayant les paramètres $\mathtt{L} = 2$ et $\mathtt{M} = 2$, et au regard du SR-composant ayant les paramètres $\mathtt{L} = 2$ et $\mathtt{M} = 1$ nous le désignons comme étant un modèle virtuel qui sera utilisé pour effectuer la composition.

Plus de notations. Avant de donner la définition de la composition sûre de SR-modèles, introduisons d'abord d'autres notations. Considérons un composant ayant les paramètres $\mathtt{M}$ et $\mathtt{L}$, et $\mathcal{A}$ le SR-modèle associé. Notons par $\mathcal{A}^{+\mathtt{m}}$ un modèle virtuel associé, obtenu par extension de sa capacité de stockage d'une quantité égale à $\mathtt{m}$; la capacité mémoire de $\mathcal{A}^{+\mathtt{m}}$ est alors $\mathtt{M}+\mathtt{m}$. Nous utiliserons un opérateur de priorité noté $\Downarrow$ pour fixer un ordre priorité dans l'application de règles : la notation $\begin{matrix} R_1 \\ \Downarrow \\ R_2 \end{matrix}$ signifie si la règle R_1 est applicable alors elle est appliquée, sinon c'est la règle R_2 qui est appliquée. Nous définissons le prédicat *exceed* qui pour un état et un entier donné indique si la quantité d'informations consommées par le composant à cet état est supérieure à cet entier ou non :

$$exceed(s, \mathtt{N}) \Leftrightarrow length(s) > \mathtt{N}$$

Naturellement, aucun état du SR-modèle associé à un SR-composant défini par des paramètres $\mathtt{L}$ et $\mathtt{M}$ ne devrait satisfaire *exceed*$(s, \mathtt{M})$. En revanche, un état d'un SR-modèle virtuel associé peut satisfaire ce prédicat puisqu'il étend la mémoire. Nous caractérisons un état d'un composant qui est prêt à produire une sortie sans se vider complètement par le prédicat *idle*:

$$idle(s) \Leftrightarrow s \xrightarrow{\bar{i}/\bar{o}} s \wedge \exists s' \in S.s \xrightarrow{\bar{i}/o} s' \wedge s' \neq s_0$$

Précisément, un état satisfaisant le prédicat *idle* est un état disposant d'au moins deux données. En effet, d'après la sémantique d'un SR-modèle, un composant ne peut rester oisif, i.e., boucler sur lui-même avec l'action $\bar{i}/\bar{o}$, que s'il a terminé de compter.

Nous définissons maintenant une nouvelle opération de composition des I/O-automates (Définition 4), cette opération permet la composition contrainte d'automates.

Définition 4 (Composition contrainte des I/O-automates)**.** *Étant donnés deux I/O-automates* $\mathcal{A}_1 = \langle S_1, s_{0_1}, \Sigma, \rightarrow_1 \rangle$ *et* $\mathcal{A}_2 = \langle S_2, s_{0_2}, \Sigma, \rightarrow_2 \rangle$ *et trois constantes* $\mathtt{l}_1$, $\mathtt{l}_2$, *et* $\mathtt{N}$, *la composition contrainte, notée* $\mathcal{A}_1 \prod_{(\mathtt{l}_1,\mathtt{l}_2,\mathtt{N})} \mathcal{A}_2$, *est un I/O-automate* $\langle S_1 \times S_2, (s_{0_1}, s_{0_2}), \Sigma, \rightarrow \rangle$ *telle que la fonction de transition* $\rightarrow$ *est la plus petite relation satisfaisant les règles suivantes:*

1. $s_1 \xrightarrow{i/\alpha^*} s_1', s_2 \xrightarrow{\beta^*/o} s_2'$

$$\begin{cases} s_1 \xrightarrow{i/o} s_1', s_2 \xrightarrow{i/o} s_2' \\ \quad \Downarrow \\ s_1 \xrightarrow{i/\bar{o}} s_1', s_2 \xrightarrow{\bar{i}/o} s_2' \end{cases} \qquad (s_1, s_2) \xrightarrow{i/o} (s_1', s_2')$$

2. $s_1 \xrightarrow{\bar{i}/\alpha^*} s_1', s_2 \xrightarrow{\beta^*/o} s_2'$

$$\begin{cases} s_1 \xrightarrow{\bar{i}/o} s_1', s_2 \xrightarrow{i/o} s_2' \quad \wedge \neg(\mathit{length}(s_1') < \mathtt{l}_1 \wedge \mathit{length}(s_2') > \mathtt{l}_2) \\ \quad \Downarrow \\ s_1 \xrightarrow{\bar{i}/\bar{o}} s_1', s_2 \xrightarrow{\bar{i}/o} s_2' \end{cases} \qquad (s_1, s_2) \xrightarrow{\bar{i}/o} (s_1', s_2')$$

3. $s_1 \xrightarrow{i/\alpha^*} s_1', s_2 \xrightarrow{\beta^*/\bar{o}} s_2'$

si $\neg\, \mathit{exceed}((s_1, s_2), \mathtt{N})$ *alors*

$$\begin{cases} s_1 \xrightarrow{i/o} s_1', s_2 \xrightarrow{i/\bar{o}} s_2' \quad \wedge \neg(\mathit{length}(s_1') < \mathtt{l}_1 \wedge \mathit{length}(s_2') > \mathtt{l}_2) \\ \quad \Downarrow \\ s_1 \xrightarrow{i/\bar{o}} s_1', s_2 \xrightarrow{\bar{i}/\bar{o}} s_2' \end{cases} \qquad (s_1, s_2) \xrightarrow{i/\bar{o}} (s_1', s_2')$$

4. $s_1 \xrightarrow{\bar{i}/\alpha^*} s_1', s_2 \xrightarrow{\beta^*/\bar{o}} s_2'$

si $\mathit{idle}(s_2)$ *alors* $\quad (s_1, s_2) \xrightarrow{\bar{i}/\bar{o}} (s_1, s_2)$

sinon

$$\begin{cases} s_1 \xrightarrow{\bar{i}/o} s_1', s_2 \xrightarrow{i/\bar{o}} s_2' \quad \wedge \mathit{length}(s_2) < \mathtt{l}_2 \\ \quad \Downarrow \\ s_1 \xrightarrow{\bar{i}/\bar{o}} s_1', s_2 \xrightarrow{\bar{i}/\bar{o}} s_2' \end{cases} \qquad (s_1, s_2) \xrightarrow{\bar{i}/\bar{o}} (s_1', s_2')$$

La composition contrainte des I/O-automates est fondée sur le principe de communication par poignée de main (handshake), seulement celle-ci est guidée par des paramètres (capacité des composants). En effet, remarquons l'utilisation de l'opérateur de priorité dans les règles qui empêche l'application simultanée des règles : dans toutes les règles le transfert est privilégié sur la pause. Le transfert d'un composant vers un autre est possible que si l'état composite résultant garantit le respect de l'ordre chronologique de toutes les entrées ($\neg(\mathit{length}(s'_1) < \mathtt{l}_1 \wedge \mathit{length}(s'_2) > \mathtt{l}_2)$). De plus, si le second composant est dans un état qui ne peut délivrer de sortie et le premier est prêt à consommer une entrée, la composition se fait qu'à condition que la quantité de données dans l'état composite n'excède pas une quantité autorisée ($\neg\ \mathit{exceed}((s_1, s_2), \mathtt{N})$). Et enfin, dans le cas où il n'y a aucune interaction des composants avec l'extérieur, si le second composant est dans un état oisif, le composant global sera alors dans un état oisif.

À présent, nous proposons une définition d'une opération de composition sûre de SR-modèles. Cette définition s'appuie sur la notion de SR-modèles virtuels qui étendent la capacité mémoire des modèles pour assurer la construction des états manquants et utilise l'opération de composition contrainte pour empêcher la construction des états indésirables.

Définition 5 (Composition des SR-modèles)**.** *Étant donnés deux composants C_1 et C_2 ayant resp. les paramètres $\mathtt{M}_1$, $\mathtt{L}_1$ et $\mathtt{M}_2$, $\mathtt{L}_2$. Le SR-modèle associé au SR-composite est obtenu par la composition dirigée des SR-modèles virtuels associés, $\mathcal{A}_1^{+\mathtt{m}_1} \prod_{(\mathtt{L}_1,\mathtt{L}_2,\mathtt{M}_1+\mathtt{M}_2)} \mathcal{A}_2^{+\mathtt{m}_2}$ avec*

$$\mathtt{m}_1 = \begin{cases} \mathit{min}(\mathtt{L}_1 - \mathtt{M}_1, \mathtt{M}_2) & \textit{si } \mathtt{M}_1 < \mathtt{L}_1 \\ 0 & \textit{sinon} \end{cases} \quad \textit{et} \quad \mathtt{m}_2 = \begin{cases} \mathtt{L}_2 - \mathtt{M}_2 & \textit{si } \mathtt{M}_2 < \mathtt{L}_2 \\ 0 & \textit{sinon} \end{cases}$$

En réalité, en utilisant l'opérateur de composition des I/O-automates, la composition de SR-modèles n'est pas correct que dans les cas où $\mathtt{M}_1 < \mathtt{L}_1$ ou $\mathtt{M}_2 < \mathtt{L}_2$. La composition dans ces cas génère des états encodant des configurations dans lesquels le système global n'est pas en mesure d'accueillir de nouvelles entrées malgré la disponibilité d'espace libre global ou il présente des irrégularités dans la progression des données. La nouvelle composition opère sur des modèles qui étendent la mémoire de composants par une quantité au moins égale à $\mathtt{L} - \mathtt{M}$ pour assurer une progression continue de données de C_1 vers C_2. En effet, elle étend la mémoire de C_1 par une quantité de données $\mathtt{m}_1$ qui permettra de recevoir les données ne pouvant être consommées en raison du mode de fonctionnement de l'ensemble du système en mode pipeline (C_1 étant plein mais C_2 disposant d'espace libre). Elle étend également la mémoire de C_2 par une quantité $\mathtt{m}_2$ pour recevoir les données que C_1 produit, mais que C_2 ne peut pas recevoir en ce moment pour cause de manque d'espace mémoire. Notons que l'utilisation de la composition contrainte garantit qu'à chaque état de l'automate résultat la quantité maximale de données consommées n'excède jamais la taille du modèle de base, i.e, $\mathtt{M}_1 + \mathtt{M}_2$.

La composition de deux SR-modèles est réalisée par le parcours des états des deux SR-modèles en partant des états initiaux. La Figure 4 illustre un exemple de SR-modèle obtenu par la composition des SR-modèles données dans la Figure 3. Notons que dans le SR-modèle résultat chaque état est codé en couleur noire par le couple d'états ayant servi à sa construction, et par dessous en couleur bleue par sa représentation dans le SR-modèle attendu. Remarquons que – le SR-modèle est déterministe : grâce à l'utilisation de l'opérateur de priorité on ne peut générer à partir d'un même état une transition étiquetée α/β et allant vers deux états différents ; – Les compositions des états ne sont pas toutes générées, par exemple l'état $\langle 1, 1\rangle\langle\rangle$ n'est pas généré. La génération de cet état résulterait de la construction de la transition suivante : $\langle 1\rangle\langle\rangle \xrightarrow{i/\bar{o}} \langle 1; 1\rangle\langle\rangle$ (par application de la règle (3)) ; Cependant, selon la règle de priorité c'est plutôt la transition $\langle 1\rangle\langle\rangle \xrightarrow{i/\bar{o}} \langle 1\rangle\langle 1\rangle$ qui est construite et l'état $\langle 1\rangle\langle 1\rangle$ qui sera alors généré. Il

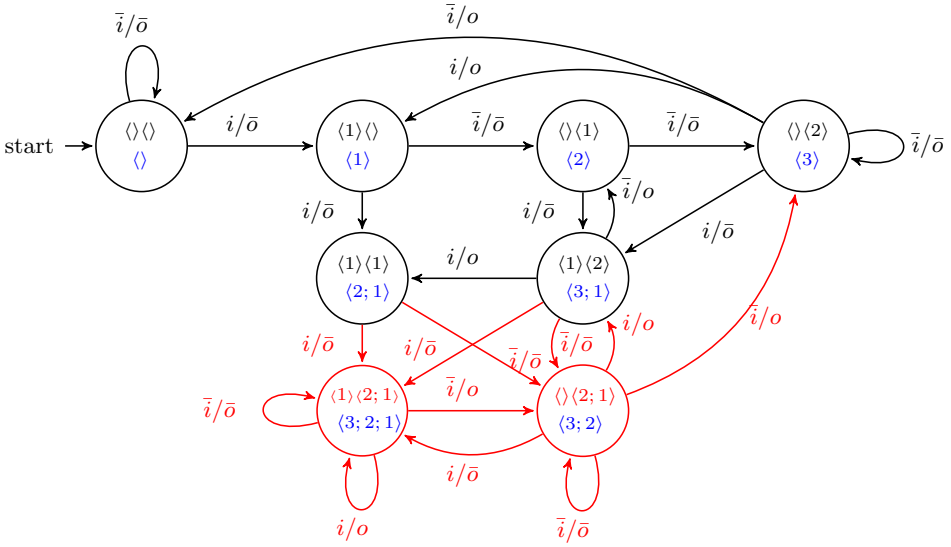

Figure 4: Le SR-modèle résultant de la composition des SR-modèles donnés dans la Figure 3

convient de noter que ces deux états sont $\mathcal{L}$-équivalent vis à vis de la latence global $\mathtt{L} = 3$, i.e., $\langle 1\rangle\langle 1\rangle \equiv_3 \langle 1;1\rangle\langle\rangle$. De manière similaire, l'état $\langle 1;1\rangle\langle 2\rangle$ n'est pas généré, on a $\langle 1\rangle\langle 2;1\rangle \equiv_3 \langle 1;1\rangle\langle 2\rangle$.

Notons que les états composites $\langle 1;1\rangle\langle 1\rangle$ et $\langle 1;1\rangle\langle 2;1\rangle$ bien que valides ils ne sont pas génères car il ne sont pas corrects. Notons aussi, compte tenu de la présence de boucles $\bar{i}/\bar{o}$ sur les états $\langle 2\rangle$ et $\langle 1;2\rangle$, l'ajout systématique des boucles $\bar{i}/\bar{o}$ sur les états composites $\langle\rangle\langle 2\rangle$ et $\langle\rangle\langle 2;1\rangle$ (application de la règle 4).

4 Validation

Dans le but de valider notre approche, nous avons implémenté en OCAML l'algorithme de construction d'un SR-modèle à partir des paramètres `M` et `L` d'un composant (appelée `build_prim`), ainsi que l'algorithme de composition permettant de construire un SR-modèle global à partir d'un ensemble de SR-modèles primitifs basé sur les règles proposées (appelée `build_comp`). Nous avons montré en utilisant l'outil de preuve Easycrypt [4] que les I/O-automates (primitifs et composites) générés par nos algorithmes satisfont bien les propriétés comportementales, i.e, les algorithmes génèrent des SR-modèles bien formés. Une description complète de l'implémentation de la construction des SR-modèles et les expérimentations menées sont disponibles dans [1].

D'autre part, nous avons utilisé l'outil de model-checking CADP [11] pour vérifier sur plusieurs exemples que l'opérateur de composition proposé préserve la sémantique des SR-systèmes. Plus précisément, nous avons montré le Lemme 1 qui dit que pour tous SR-composants (caractérisés par leurs paramètres $\mathtt{L}_1$, $\mathtt{M}_1$, $\mathtt{L}_2$, et $\mathtt{M}_2$), le modèle obtenu par la composition des modèles associés est bisimilaire au modèle associé au SR-composant composite (caractérisé par $\mathtt{L}_1 + \mathtt{L}_2$, $\mathtt{M}_1 + \mathtt{M}_2$).

Lemma 1. *Pour tous entiers* $\mathtt{M}_1, \mathtt{L}_1, \mathtt{M}_2, \mathtt{L}_2 \in \mathbb{N}$,
`build-prim`$(\mathtt{M}_1 + \mathtt{M}_2, \mathtt{L}_1 + \mathtt{L}_2)$ *et* *`build-comp`*$(\mathtt{M}_1, \mathtt{L}_1, \mathtt{M}_2, \mathtt{L}_2)$ *sont bisimilaires.*

Nous avons aussi mené l'expérimentation sur une étude de cas réelle, une application de traitement de signal présentée dans [17] : application IDCT (*Inverse discrete cosine transform*) qui comporte sept composants connectés en série et qui communiquent entre eux selon le principe de la poignée de main. L'analyse et la vérification ont pris quelques millisecondes. Nous avons utilisé les SR-modèles générés pour vérifier la satisfiabilité de propriétés comportementales. Les propriétés sont alors écrites en langage MCL (*Model Checking Language*) [18], langage supporté par CADP. Les formules MCL sont des formules logiques basées sur des expressions régulières utilisant des opérateurs booléens, des opérateurs de modalité (l'opérateur de nécessité est noté par les crochets [ ], et l'opérateur de possibilité est noté par des chevrons ⟨ ⟩), l'opérateur de point fixe maximal et des constructions de traitement de données inspirées des langages de programmation fonctionnels. Pour les machines à états, une séquence de transitions qui commence par l'état initial et qui satisfait une formule régulière φ peut être exprimé en MCL, soit comme exemple par la formule $\langle\varphi\rangle$**true**, soit comme contre-exemple par la formule $[\varphi]$**false**. Les séquences de transition sont spécifiées en MCL par des formules d'actions et des opérateurs d'expressions régulières. Pour un SR-modèle associé à un SR-composant caractérisé par `L` et `M` nous avons vérifié par exemple :

- La propriété exprimant qu'après la lecture d'une information le composant devra attendre $\mathtt{L}-1$ unité de temps (en plus de l'unité de temps consacré à la consommation) avant de produire la sortie correspondante, est codée par la formule MCL suivante :

  $[(\neg\ \text{"}\mathtt{i}/\bar{o}\text{"})*.\text{"}\mathtt{i}/\bar{o}\text{"}]\ <\mathtt{true}\{\mathtt{L}-1\}.(\text{"}\mathtt{i/o}\text{"} \vee \text{"}\bar{i}/\mathtt{o}\text{"})>\ \mathtt{true}$

  L'expression $\mathtt{true}\{\mathtt{L}–1\}$ représente les séquences de transitions qui contiennent exactement $\mathtt{L}-1$ occurrences de n'importe quelle action (encodée en MCL par la constante `true`).
- De la même manière, la propriété exprimant que le composant peut recevoir jusqu'à `M` données, après cela, il ne peut plus recevoir de données se code par la formule :

  $[(\neg\ \text{"}\mathtt{i}/\bar{o}\text{"})*.(\text{"}\mathtt{i}/\bar{o}\text{"})\{\mathtt{M}\}]\ <(\neg\ \text{"}i/\bar{o}\text{"})>\ \mathtt{true}$

Sans aucune surprise ces propriétés sont évaluées automatiquement par l'outil à true (vrai) ce qui signifie que les SR-modèles respectent la sémantique des SR-systèmes. De la même manière, nous pouvons coder des propriétés de sûreté et les vérifier sur les modèles ainsi générés.

5 Travaux connexes

Le développement de systèmes à base de composants est une approche qui a fait ces preuves dans plusieurs domaines, particulièrement, dans les systèmes embarqués et distribués. Il existe un grand nombre de modèles de composants qui sont utilisés pour la conception de systèmes matériels ou logiciels complexes, citons : Ptolemy [9], BIP [5] et GCM [2]. En revanche, il existe peu de travaux sur les théories permettant de raisonner sur la compositionnalité, spécialement sur la modélisation formelle et la vérification des systèmes construits à base de composants. Parmi les recherches dédiées à la vérification comportementale orientée composants dont on a connaissance, on peut citer Kmelia [3], STSLib [10] et BIP. Bien que ces approches proposent des outils de conception permettant le raisonnement compositionnel sur les systèmes embarqués et distribués, ils ne sont pas dédiés à l'analyse des applications réactives. Le travail le plus proche de notre approche est BIP, l'outil BIP fournit une approche formelle pour la modélisation du comportement et l'architecture des systèmes. Bien que des travaux aient été menés [6] pour étendre les modèles BIP et permettre l'analyse de systèmes synchrones, il a été démontré que la sémantique proposée n'est pas complète et la synchronie est affaiblie dans ce formalisme.

Du point de vue de la théorie des I/O-automates, outre les travaux autour de la modélisation des systèmes à base d'événements nous citons [19], les auteurs montrent sur l'exemple du protocole bit alterné comment grâce à techniques de preuve par déduction peuvent être construits des modèles finis de systèmes infinis, ces modèles finis qui sont des I/O-automates sont utilisés pour faire de la preuve de correction par model-checking.

Par ailleurs, dans la perspective de modélisation des systèmes de type dataflow, dans la littérature il existe un grand nombre d'approches pour la spécification et la conception de ces systèmes. Parmi les plus utilisés, citons les réseaux de Kahn [12], les graphes synchrones de flot de données (SDFGs) [15]. Pour la plupart leur modèle de calcul est un graphe orienté coordonné ou pas par une horloge. Dans [8] les auteurs montrent que le manque d'horloge rend les modèles inadaptés au raisonnement formel et à la génération de codes synchrones corrects-par-construction. Notre approche s'inscrit dans la démarche dite conduite par le temps, e.g., [13] dans laquelle la production des événements et leur prise en compte par le système se fait à des instants connus sous l'hypothèse de synchronie. Ainsi la date de production des données est parfaitement connue et ne dépend pas de la durée effective du traitement.

6 Conclusion

Dans ce papier, nous avons présenté des modèles pour la modélisation des systèmes synchrones et des règles de composition de ces modèles qui permettent de raisonner sur des systèmes complexes, obtenus par assemblage en série de composants synchrones. Ces modèles sont basées sur les I/O-automates. qui ont un intérêt plus général d'un point de vue de l'ingénierie, ils permettent de raisonner sur des systèmes basés composants de différents types : composants réels, logiciels, canaux de communication, capteurs, etc. De manière générale, notre solution gère la composition et le traitement de flux entre les composants. Les algorithmes de construction des modèles proposées ont été implémentés et leur validité montrée par l'utilisation d'outils de model checking CADP. Compte tenu de la contrainte que tous les états d'un I/O-automate ne sont pas valides, les modèles générés ne souffrent pas du problème spécifique à la composition d'automates qui est l'explosion d'états. Par exemple, la composition de deux composants C_1 et C_2 avec les paramètres $\mathtt{M}_1 = 6$, $\mathtt{L}_1 = 9$ et $\mathtt{M}_2 = 3$, $\mathtt{L}_2 = 6$ génère 27824 états et 76465 transitions.

Une suite directe à ce travail est de s'attaquer à l'opérateur de composition en parallèle, ce qui permettra de modéliser et d'analyser une classe plus large d'applications. Aussi étendre la sémantique des SR-modèles à d'autres relations entre les entrées et les sorties. En effet, jusqu'ici nous avons considéré des composants qui à chaque instant la sortie dépend d'une seule entrée et aussi les entrées et sorties sont de largeur 1. Pour étendre la famille de composants analysés, il faut relâcher cette hypothèse et considérer des modèles pouvant supporter des relations de type plusieurs entrées/une sortie et plusieurs entrées/plusieurs sorties. Une autre piste qui serait intéressante à creuser est d'étudier d'autres types composants ayant d'autres caractéristiques.

References

[1] Correct-by-construction SR-systems. https://github.com/SarahChabane/correct-by-construction-SR-systems.git.

[2] R. Ameur-Boulifa, L. Henrio, O. Kulankhina, E. Madelaine, and A. Savu. Behavioural semantics for asynchronous components. *Journal of Logical and Algebraic Methods in Programming*, 89:1 – 40, 2017.

[3] Pascal André, Gilles Ardourel, and Christian Attiogbé. Composing Components with Shared Services in the Kmelia Model. In *7th International Symposium on Software Composition, SC'08*, volume 4954 of *LNCS*. Springer, 2008.

[4] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. Easycrypt: A tutorial. In *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures*, pages 146–166, 2013.

[5] Ananda Basu, Bensalem Bensalem, Marius Bozga, Jacques Combaz, Mohamad Jaber, Thanh-Hung Nguyen, and Joseph Sifakis. Rigorous Component-Based System Design Using the BIP Framework. *IEEE Software*, 28(3):41–48, May 2011.

[6] Marius Dorel Bozga, Vassiliki Sfyrla, and Joseph Sifakis. Modeling Synchronous Systems in BIP. In *Proceedings of the Seventh ACM International Conference on Embedded Software*, EMSOFT '09, pages 77–86, New York, NY, USA, 2009. ACM.

[7] Sarah Chabane, Rabea Ameur-Boulifa, and Mohamed Mezghiche. Rethinking of i/o-automata composition. In *Forum on Specification and Design Languages (FDL), 2017*, pages 1–7. IEEE, 2017.

[8] Albert Cohen, Marc Duranton, Christine Eisenbeis, Claire Pagetti, Florence Plateau, and Marc Pouzet. Synchronization of periodic clocks. In *Proceedings of the 5th ACM international conference on Embedded software*, pages 339–342. ACM, 2005.

[9] Johan Eker, Jörn W. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Stephen Neuendorffer, Sonia R. Sachs, and Yuhong Xiong. Taming Heterogeneity - the Ptolemy Approach. *Proceedings of the IEEE*, 91(1):127–144, January 2003.

[10] Fabrício Fernandes and Jean-Claude Royer. The STSLib project: Towards a formal component model based on STS. *Electronic Notes in Theoretical Computer Science*, 215:131–149, 2008.

[11] H. Garavel, F. Lang, and R. Mateescu. An overview of CADP 2001. *European Association for Software Science and Technology Newsletter*, 4:13–24, aug 2002.

[12] KAHN Gilles. The semantics of a simple language for parallel programming. *Information processing*, 74:471–475, 1974.

[13] Thomas A. Henzinger, Benjamin Horowitz, and Christoph M. Kirsch. Giotto: a time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84–99, 2003.

[14] Hermann Kopetz, Martin Braun, Christian Ebner, Andreas Krüger, Dietmar Millinger, Roman Nossal, and Anton V. Schedl. The design of large real-time systems: The time-triggered approach. In *16th IEEE Real-Time Systems Symposium, Palazzo dei Congressi, Via Matteotti, 1, Pisa, Italy, December 4-7, 1995, Proceedings*, pages 182–187, 1995.

[15] Edward Ashford Lee and David G Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on computers*, 100(1):24–35, 1987.

[16] N. Lynch and M. Tuttle. An introduction to Input/Output automata. *CWI-Quarterkly*, 2(3):219–246, 1989.

[17] R. Pacalet M. Baclet and A. Petit. Register transfer level simulation. *Research Report LSV-04-10. Laboratoire Spécification et Vérification. ENS de Cachan. France*, may, 2004.

[18] Radu Mateescu and Damien Thivolle. A model checking language for concurrent value-passing systems. In *FM 2008: Formal Methods, 15th International Symposium on Formal Methods, Turku, Finland, May 26-30, 2008, Proceedings*, pages 148–164, 2008.

[19] Olaf Müller and Tobias Nipkow. Combining model checking and deduction for i/o-automata. In *International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, pages 1–16. Springer, 1995.

[20] Joseph Sifakis. A framework for component-based construction extended abstract. In *Third IEEE International Conference on Software Engineering and Formal Methods (SEFM 2005), 7-9 September 2005, Koblenz, Germany*, pages 293–300, 2005.

A Algorithme de construction des SR-modèles

Nous introduisons d'abord les opérateurs et fonctions sur les listes qui sont utilisés dans l'algorithme 1.

- L'opérateur $\lhd$: $s \lhd i$ efface l'indice i de la liste s;
- L'opérateur $\uplus$: $s \uplus i$ étend s avec un nouvel indice i.
- L'opérateur $\oplus$: $s \oplus l$ incrémente les valeurs des indices de s. L'incrémentation est bornée par la valeur l (si s est vide alors $s \oplus l$ retourne le tuple vide).
- La fonction *fresh*(s) crée un nouvel état correspondant à la liste s.
- La fonction *CreateTransition*(s, α, s') : crée et retourne la transition $s \xrightarrow{\alpha} s'$.
- La fonction *greatest*(s) renvoie la date la plus grande de la liste - correspond à la fonction *lfst*(s).
- La fonction $greatest_2(s)$ renvoie la deuxième date la plus grande de la liste.
- La fonction *Exists.equiv*(s, S) vérifie s'il existe un état équivalent de s dans S.
- La fonction *S.equiv*(s) retourne l'état équivalent à s dans S.

Algorithme 1 : SR-Model($\mathtt{M}, \mathtt{L}$)

$s_0 \leftarrow fresh(\langle\rangle)$;
States $\leftarrow \{s_0\}$;
Transitions $\leftarrow \{\}$;
Transitions $\leftarrow$ Transitions $\cup$ $CreateTransition(s_0, \bar{i}/\bar{o}, s_0)$;
$s_1 \leftarrow fresh((s_0 \oplus \mathtt{L}) \uplus 1)$;
Transitions $\leftarrow$ Transitions $\cup$ $CreateTransition(s_0, i/\bar{o}, s_1)$;
Unexplored $\leftarrow \{s_1\}$;
States $\leftarrow$ States $\cup \{s_1\}$;
while Unexplored $\neq \{\}$ **do**
  take s in Unexplored;
  if *length(s)* $< \mathtt{M}$ **then**
    $s_1 \leftarrow fresh((s \oplus \mathtt{L}) \uplus 1)$;
    if $Exists.equiv(s_1$,States$)$ **then** $s_2 \leftarrow$States.$equiv(s_1)$;
    else $s_2 \leftarrow s_1$;
    Transitions $\leftarrow$ Transitions $\cup$ $CreateTransition(s, i/\bar{o}, s_2)$;
    if $s_2 \notin$ States **then**
      Unexplored$\leftarrow$Unexplored $\cup \{s_2\}$; States$\leftarrow$States $\cup \{s_2\}$;
  if *greatest(s)* $= \mathtt{L}$ **then**
    $s_1 \leftarrow fresh(((s \lhd greatest(s)) \oplus \mathtt{L}) \uplus 1)$;
    if $Exists.equiv(s_1$,States$)$ **then** $s'_1 \leftarrow$States.$equiv(s_1)$;
    else $s'_1 \leftarrow s_1$;
    Transitions $\leftarrow$ Transitions $\cup$ $CreateTransition(s, i/o, s'_1)$;
    if $s'_1 \notin$ States **then**
      Unexplored$\leftarrow$Unexplored $\cup \{s'_1\}$; States$\leftarrow$States $\cup \{s'_1\}$;
    $s_2 \leftarrow fresh((s \lhd greatest(s)) \oplus \mathtt{L})$;
    if $Exists.equiv(s_2$,States$)$ **then** $s'_2 \leftarrow$States.$equiv(s_2)$;
    else $s'_2 \leftarrow s_2$;
    Transitions $\leftarrow$ Transitions $\cup$ $CreateTransition(s, \bar{i}/o, s'_2)$;
    if $s'_2 \notin$ States **then**
      Unexplored$\leftarrow$Unexplored $\cup \{s'_2\}$; States$\leftarrow$States $\cup \{s'_2\}$;
    if $length(s) \geq 2 \wedge (greatest(s) - greatest_2(s)) > 1$ **then**
      $s_1 \leftarrow fresh(s \oplus \mathtt{L})$;
      if $Exists.equiv(s_1$,States$)$ **then** $s_2 \leftarrow$States.$equiv(s_1)$;
      else $s_2 \leftarrow s_1$;
      Transitions $\leftarrow$ Transitions $\cup$ $CreateTransition(s, \bar{i}/\bar{o}, s_2)$;
      if $s_2 \notin$ States **then**
        Unexplored$\leftarrow$Unexplored $\cup \{s_2\}$; States$\leftarrow$States $\cup \{s_2\}$;
    else
      Transitions $\leftarrow$ Transitions $\cup$ $CreateTransition(s, \bar{i}/\bar{o}, s)$;
  if $s \neq s_0 \wedge greatest(s) < \mathtt{L}$ **then**
    $s_1 \leftarrow fresh(s \oplus \mathtt{L})$;
    if $Exists.equiv(s_1$,States$)$ **then** $s_2 \leftarrow$States.$equiv(s_1)$;
    else $s_2 \leftarrow s_1$;
    Transitions $\leftarrow$ Transitions $\cup$ $CreateTransition(s, \bar{i}/\bar{o}, s_2)$;
    if $s_2 \notin$ States **then**
      Unexplored$\leftarrow$Unexplored $\cup \{s_2\}$; States$\leftarrow$States $\cup \{s_2\}$;
  Unexplored $\leftarrow$ Unexplored$-\{s\}$;
return (States,s_0,Transitions);