Vérifications du protocole VCI[1]

Manuel Baclet* — Renaud Pacalet**

* *LSV - CNRS UMR 8643 - ENS de Cachan*
61, Av. du Président Wilson - F-94235 Cachan Cedex

baclet@lsv.ens-cachan.fr

** *ENST / COMELEC*
46 rue Barrault - F-75634 Paris Cedex 13

pacalet@enst.fr

RÉSUMÉ. *Cet article présente une norme de communication (la norme VCI) entre composants virtuels ainsi qu'une méthode de vérification formelle permettant d'assurer que des implémentations du protocole sont conformes aux spécifications de la norme. Nous décrivons ensuite les résultats obtenus avec le model-checker symbolique NuSMV.*

ABSTRACT. *This paper describes a communication standard (VCI) between virtual components and a formal method of verification that allows to check whether implementations of the protocol follow the specifications of the standard or not. Finally, we detail our results, using the symbolic model-checker NuSMV.*

MOTS-CLÉS : *VCI, composants virtuels, model-checking, systèmes sur puce, NuSMV.*

KEYWORDS: *VCI, virtual components, model-checking, systems on chip, NuSMV.*

1. Travail soutenu par l'AS 55 (Méthodes formelles de vérification) du Réseau Thématique Pluridisciplinaire SOC.

1. Introduction

L'industrie du semi-conducteur est confrontée à une difficulté récurrente : la relative inadéquation des outils de conception à la complexité des dispositifs créés. En effet, les progrès considérables accomplis année après année par les technologues permettent d'intégrer sur un même substrat de silicium toujours plus de fonctions. On conçoit aujourd'hui des systèmes intégrés (ou SoC, pour System on Chip) réunissant plusieurs millions de transistors. Ces systèmes embarquent fréquemment plusieurs micro-processeurs, des fonctions de traitement analogique ou numérique du signal, des interfaces multiples avec le monde extérieur, etc.

Les outils et méthodes utilisés pour la conception de tels dispositifs sont, pour l'essentiel, ceux que les concepteurs utilisaient il y a peu pour concevoir des circuits spécifiques (ASIC pour Application Specific Integrated Circuits) beaucoup plus simples. L'inadéquation apparaît principalement dans la baisse de la productivité relative des concepteurs et dans la proportion croissante du temps de conception consacré à la vérification (essentiellement de la simulation, actuellement).

Plusieurs pistes existent qui contribuent à résoudre ces difficultés :

– le recours à des méthodes de vérification complémentaires de la simulation, les méthodes formelles, qui permettent une vérification exhaustive de tous les comportements d'un système, moyennant éventuellement quelques simplifications (abstractions).

– la réutilisation d'éléments complexes (les composants virtuels ou IP, pour Intellectual Property) dans des projets successifs.

Pour réutiliser plus facilement un composant virtuel, comme un cœur de microprocesseur ou une fonction de traitement du signal, il est indispensable que son interface soit normalisée. Il est ainsi inutile de modifier le composant lorsque change son environnement. Cette stratégie fait l'objet d'une promotion active par un consortium international d'industriels et de laboratoires de recherche, VSIA (Virtual Socket Interface Alliance, http://www.vsi.org/) qui a produit une norme définissant l'interface des composants virtuels, la norme VCI (Virtual Component Interface).

Un composant respectant la norme VCI présentera donc l'avantage inestimable de s'interfacer simplement et sans développements supplémentaires avec une architecture de système intégré utilisant cette même norme. La question se pose donc inévitablement de qualifier les composants réutilisables et de vérifier qu'ils respectent véritablement la norme VCI. Les travaux présentés ici s'inscrivent dans ce cadre.

Dans la section 2, nous présentons la norme VCI. Celle-ci décrit une communication point à point et asymétrique entre deux acteurs : un *initiateur* et une *cible*. Les *spécifications* de la norme sont également présentées. Notre premier objectif était de réaliser un modèle de cible générique qui permette de tester, lorsqu'on la lie à un modèle d'initiateur, si celui-ci respecte la norme VCI. La modélisation que nous avons réalisée sous forme d'automates finis à variables partagées est décrite dans la section 3. Le formalisme choisi permet une vérification exhaustive des comportements de l'initiateur. La section 4 présente (rapidement) le model-checker *NuSMV* que nous avons utilisé ainsi que les bons résultats obtenus, pour l'instant, sur des modèles d'initiateurs simples. Finalement, nous présentons les travaux futurs que nous envisageons.

2. Description de la norme VCI

La norme VCI se décline en trois versions plus ou moins élaborées selon la nature et le rôle des composants. La déclinaison Peripheral (PVCI) est destinée à des composants d'entrée - sortie. Elle est la plus simple et c'est celle que nous avons choisi d'analyser en détails. Deux versions plus performantes existent, Basic (BVCI) et Advanced (AVCI) qui feront l'objet de travaux futurs.

2.1. *L'initiateur et la cible*

La norme VCI définit une communication point à point et asymétrique entre un *initiateur* de communication et une *cible*. L'initiateur émet des requêtes (de lecture ou d´écriture, simples ou par paquets) et la cible y répond, si elle le peut.

Les signaux qui constituent une interface PVCI sont résumés dans le tableau suivant :

Nom (taille, en bits)	Origine	Description
VAL (1)	Init.	Indicateur de requête valide
RD (1)	Init.	Indicateur de lecture ou d'écriture
ADDRESS (n)	Init.	Adresse de lecture ou d'écriture
BE (b)	Init.	Byte Enable : octets concernés par la requête
WDATA (8b)	Init.	Données associées à une requête d'écriture
EOP (1)	Init.	End Of Packet signale la fin d'un paquet
ACK (1)	Cible	Indique que la requête courante est acceptée
RDATA (8b)	Cible	Données associées à une requête de lecture
RERROR (e)	Cible	Signalisation des erreurs

où $b \in \{1, 2, 4\}$, $e \in \{1, 2, 3\}$ et $n \in \{0..64\}$.

L'initiateur et la cible sont tous deux des dispositifs synchrones sur front montant d'une horloge CLK et munis d'un signal de réinitialisation synchrone RESET. Ces deux signaux sont gérés par le système d'accueil. Lorsque l'initiateur désire soumettre une requête à sa cible il active le signal VAL, positionne les signaux RD, ADDRESS, BE et WDATA et attend l'acceptation de la requête par la cible (signal ACK). Il ne peut soumettre une nouvelle requête tant que la précédente n'est pas acceptée et il ne peut modifier la requête en cours, ni même y renoncer. Il est toutefois possible pour la cible d'avoir un comportement asynchrone : celle-ci peut fournir une réponse à l'initiateur en un cycle d'horloge, comme expliqué dans la section 3.2. Les chronogrammes suivants illustrent différentes transactions possibles entre initiateur et cible.

Le chronogramme de la figure 1 décrit une séquence de cycles de lecture et d'écriture simples. Celui de la figure 2 présente une séquence où la cible traite toutes les requêtes en un cycle d'horloge.

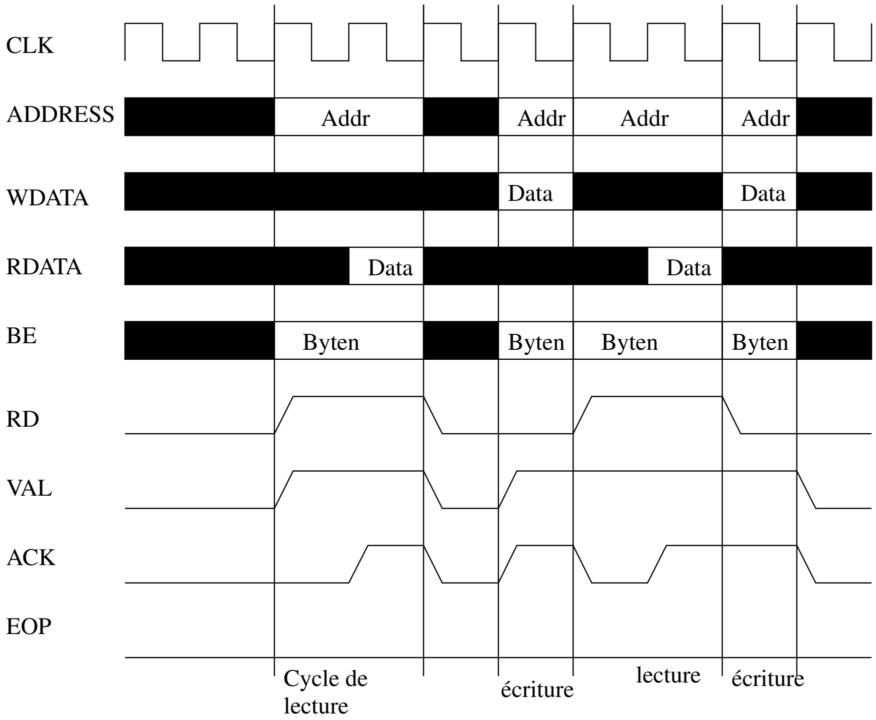

Figure 1. *Exemples de transactions*

Dans toute la suite, on suppose que b est fixé à 4, e à 3 et n à 8.

2.2. *Spécifications*

Les propriétés que doivent respecter les acteurs du protocole sont les suivantes :

– Les requêtes de l'initiateur doivent être maintenues, ce qui signifie que l'initiateur ne peut pas modifier ses sorties tant que sa requête n'a pas été acquittée par la cible.

– Une nouvelle requête qui appartient à un paquet doit avoir la même direction (lecture ou écriture) que la précédente dans le même paquet.

– Une nouvelle requête qui appartient à un paquet doit avoir l'adresse de la requête précédente plus 4.

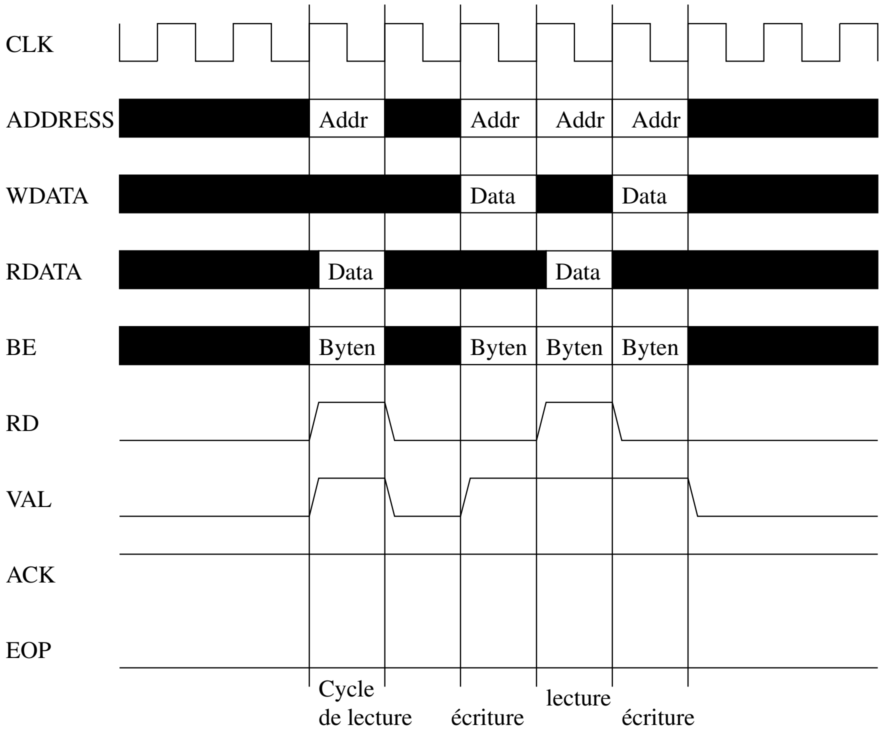

Figure 2. *Exemples de transactions*

– Une nouvelle requête qui est la première d'un paquet doit avoir pour adresse un multiple de 4.

– Une nouvelle requête doit avoir un champ BE valide : toutes les valeurs possibles ne sont pas autorisées.

Il n'y a pas de contrainte pour la cible. Par exemple, une cible qui ne répond rien respecte le protocole.

Afin de valider le plus simplement possible un initiateur PVCI nous avons conçu un modèle de cible PVCI spécifique dont le seul rôle est de répondre aux requêtes, de la façon la plus variée possible, tout en respectant la norme et en vérifiant que l'initiateur auquel elle est connectée respecte lui aussi la norme. La qualification d'un composant initiateur passe donc par sa mise en relation avec cette cible et par l'utilisation d'un model-checker qui vérifiera si la cible respecte la norme. C'est le travail décrit dans la suite.

3. Modélisation

Nous voulons construire un modèle de cible générique qui, lorsqu'on la lie à un modèle d'initiateur, fournit des réponses aux requêtes de l'initiateur et vérifie que celui-ci se comporte correctement vis-à-vis du protocole VCI. Les différents éléments du système seront modélisés par des automates finis à variables partagées.

3.1. *Automates à variables partagées*

3.1.1. *Formalisme*

On se donne un ensemble $V = \{x_1, .., x_n\}$ de variables booléennes et on note V', l'ensemble des variables $\{x'_1, .., x'_n\}$.
Une *mise à jour* est une formule booléenne sur les variables $V \cup V'$. Les booléens $vrai$ et $faux$ seront notés respectivement 1 et 0.

Un *automate à variables partagées* sur V est un triplet $\mathcal{A} = (Q, I, T)$ où :

– Q est un ensemble fini d'états.

– $I \subset Q \times Val(V)$ est l'ensemble des états initiaux. ($Val(V)$ désigne l'ensemble des valuations de V)

– $T \subset Q \times Up \times Q$ est l'ensemble des transitions, où Up désigne l'ensemble des mises à jour.

Un *état* de $\mathcal{A}$ est un couple (q, val), où $q \in Q$ et val est une valuation des variables de V.

Informellement, une transition $\tau = (q, up, q')$ permet de passer d'un état (q, val) à un état (q', val') si :

$$up(val, val') = 1, \text{ i.e. } val' \text{ est une mise à jour possible de } val.$$

Un exemple d'automate à variables partagées est présenté dans la figure 3.

3.1.2. *Produit d'automates*

Si $\mathcal{A} = (Q_1, I_1, T_1)$ et $\mathcal{B} = (Q_2, I_2, T_2)$ sont des automates à variables partagées sur $V = \{x_1, .., x_n\}$, le produit de $\mathcal{A}$ et $\mathcal{B}$, noté $\mathcal{A} \times \mathcal{B}$ est l'automate à variables partagées sur V défini par :

$$\mathcal{A} \times \mathcal{B} = (Q_1 \times Q_2, I_1 \times I_2, T)$$

où :

$$T = \{((q_1, q_2), up_1 \wedge up_2, (q'_1, q'_2)) | (q_1, up_1, q'_1) \in T_1$$
$$\text{et } (q_2, up_2, q'_2) \in T_2\}$$

L´état initial de l'automate est l'état $(ready, \neg request)$.

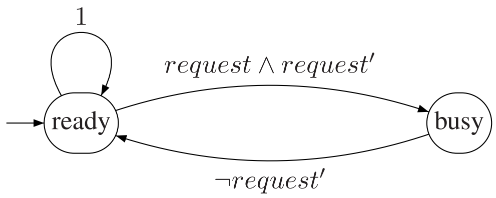

Figure 3. *Exemple d'automate à variables partagées*

3.2. *Modélisation des cycles d'horloge*

Nous allons nous arranger pour qu'un cycle d'horloge corresponde à une transition dans notre automate.

L'initiateur étant synchrone, il échantillonne ses entrées au front montant d'horloge et met à jour ses sorties au cours du cycle. On peut donc modéliser son comportement de la façon suivante :

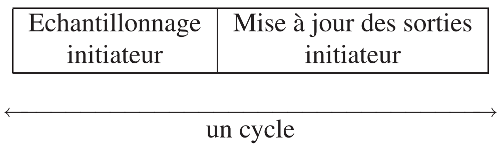

Figure 4. *Modélisation des cycles*

Le comportement de la cible est un peu plus complexe puisqu'elle peut satisfaire une requête de l'initiateur en un cycle d'horloge (réponse asynchrone), ce qui revient à dire que la cible échantillonne ses entrées avant le front montant d'horloge (mais après que l'initiateur a mis à jour ses sorties) et met à jour ses sorties avant la fin du cycle. On a donc le schéma d'horloge suivant :

Echantillonnage initiateur	Mise à jour des sorties initiateur	Echantillonnage cible	Mise à jour des sorties cible

un cycle

Figure 5. *Modélisation des cycles*

3.3. *Un exemple*

Nous présentons un système simple constitué de trois automates, dont certains seront réutilisés dans la suite :

– Un automate représentant le système et mettant à jour la variable $reset$. (reset est actif quand $reset$ vaut 0.)

– Un automate représentant la cible qui fournit des réponses aléatoires à l'initiateur.

– Un initiateur enchaînant des requêtes de lecture ou d'écriture simples et des cycles d'inactivité, aléatoirement.

Le système final est le produit de ces trois automates.

3.3.1. *Système*

L'état initial de l'automate système est défini par la formule $reset$.

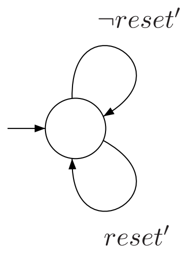

Figure 6. *Automate système*

3.3.2. *Cible*

La cible est représentée dans la figure 7. L'état initial de l'automate cible est défini par la formule $\neg ack \wedge (rerror = 000) \wedge (rdata = 0)$.

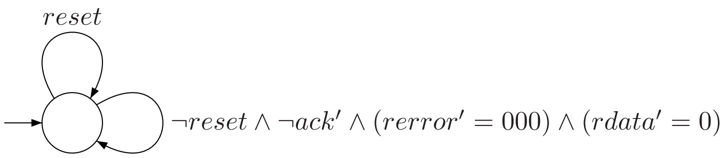

Figure 7. *Automate cible*

3.3.3. *Initiateur*

L'initiateur est représenté dans la figure 8.

Les états initiaux sont définis par la formule : $\neg val \wedge eop \wedge rd \wedge (address = 0) \wedge (be = 15) \wedge (wdata = 0)$.

La macro $Reset_I$ correspond à la réinitialisation des sorties de l'initiateur et la macro $Maintien\_req$ signifie que les valeurs des sorties restent inchangées après passage de la transition.

Nous présentons à présent notre modèle.

3.4. *Système*

L'automate représentant le système est identique à celui de l'exemple de la section 3.3. Lors de chaque transition, il donne à la variable $reset$ la valeur 0 ou 1, de façon non déterministe.

3.5. *Cible-Vérificateur*

Notre cible-vérificateur est le produit des deux automates suivants.

3.5.1. *Cible*

Notre modèle de cible sera la cible de l'exemple de la section 3.3. Les sorties de la cible sont :

– la variable booléenne ack

– la variable codée sur 32 bits $rdata$

– la variable codée sur 3 bits $rerror$ qui peut prendre les valeurs de l'ensemble $\{000, 001, 101, 111\}$

Lorsque le reset est inactif, les réponses de notre cible sont aléatoires et, en particulier, ne dépendent pas des sorties de l'initiateur. Si le reset est actif, les sorties de la

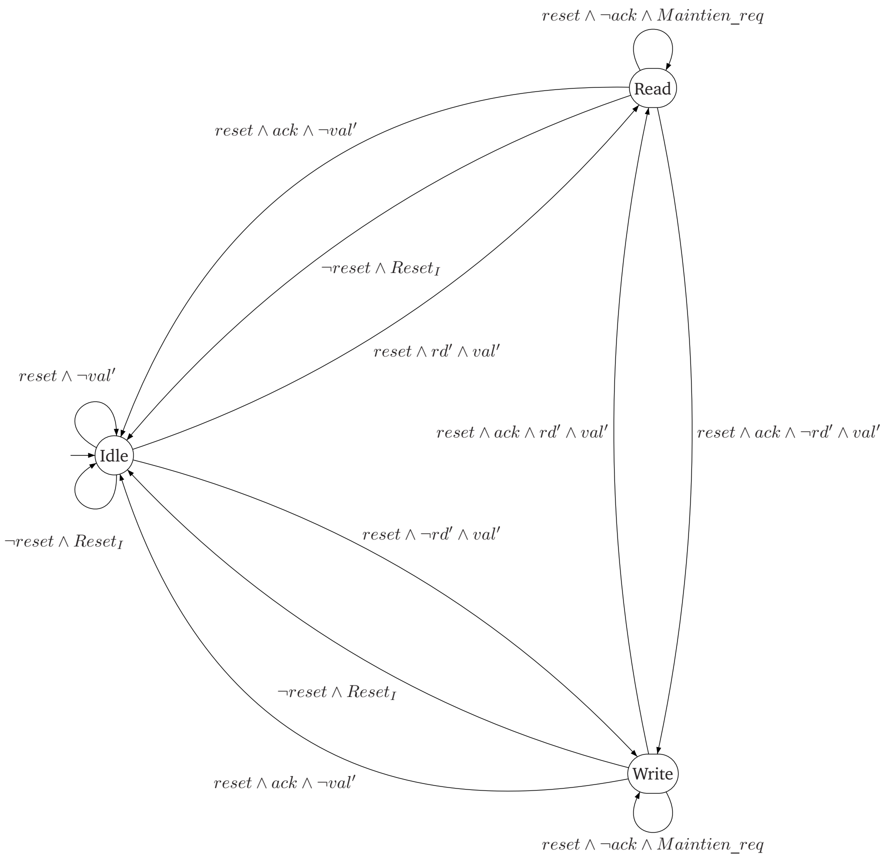

Figure 8. *Exemple d'automate initiateur*

cible sont réinitialisées de la façon suivante : ack prend la valeur 0, $rerror$ prend la valeur 000 et $rdata$ prend la valeur 0.

3.5.2. *Vérificateur*

Le vérificateur a pour tâche de vérifier que le comportement de l'initiateur est conforme à la spécification de la section 2.2.

L'automate vérificateur dispose de deux états : Courant et Erreur et de 6 variables internes : $req.val, req.eop, req.rd, req.address, req.be$ et $req.wdata$ qui codent, si elle existe, une requête en cours de l'initiateur.

Le prédicat $Erreur$ est une fonction (booléenne) des variables internes et des sorties de la cible et de l'initiateur. Il permet de détecter une faute dans le comportement de l'initiateur. La macro $Reset_V$ permet de réinitialiser les variables internes du vérificateur et la macro MaJ_V pemet de mettre à jour les variables internes en fonction des sorties de la cible et de l'initiateur.

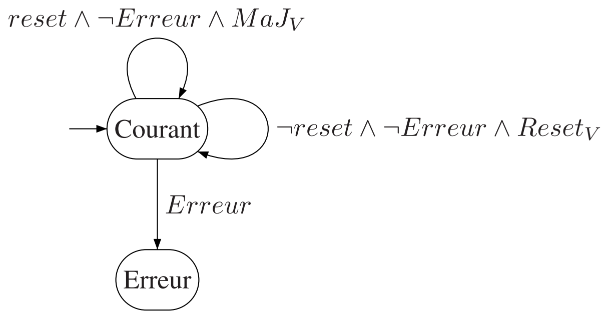

Figure 9. *Automate vérificateur*

Le fonctionnement de l'automate vérificateur est détaillé dans l'annexe.

3.6. *Initiateurs*

Nous avons analysé quatre modèles d'initiateurs :

-I1 est un initiateur similaire à celui de la figure 8 mais qui peut également effectuer des requêtes de lecture ou d'écriture par paquets.

-I2 est semblable à l'initiateur I1 mais il peut, au milieu d'une requête de lecture par paquets, lancer une requête d'écriture.

-I3 dispose de trois variables internes : $lr.rd$, $lr.eop$ et $lr.address$ qui stockent la prochaine requête.

I3 se comporte de la façon suivante :

i. Si $val \wedge ack$, la prochaine requête, qui est choisie aléatoirement, est stockée dans la structure lr, en notant que si $\neg lr.eop$, $lr.address$ est un multiple de 4 et que si on est au cours d'une opération par paquet ($\neg eop$), $lr.address$ est augmenté de 4 et $lr.rd$ reste inchangé.

ii. Si $\neg val \vee ack$, l'initiateur peut ou non émettre une nouvelle requête. Si c'est le cas, ses sortie rd, eop et $address$ sont copiées sur la structure lr. Sinon, ses sorties autres que val sont choisie aléatoirement.

iii. Si $val \wedge \neg ack$, les valeurs des variables et des sorties sont inchangées.

-I4 est similaire à l'initiateur I3. La seule différence est que dans la première ligne d'instruction, $val \wedge ack$ est remplacé par ack.

Ces deux derniers exemples montrent qu'il est difficile de lire sur la description d'un initiateur si celui-ci respecte le protocole.

4. Résultats

4.1. *NuSMV*

Nous avons décidé d'utiliser le model-checker NuSMV qui est une extension de SMV. Il est distribué sous licence OpenSource et disponible à l'adresse :

$$\mathbf{http} : //\mathbf{nusmv.irst.itc.it}$$

Les points forts de NuSMV sont :

– représentation symbolique des ensembles d'états en utilisant des diagrammes de décision binaires (ROBDD, voir, par exemple [MCM 93]) ;

– possibilité de tester des propriétés exprimable dans les logiques temporelles CTL, LTL et RTCTL (Real Time CTL) ;

– une syntaxe pour la description des systèmes (relativement) proche de celle du langage de description des composants virtuels (System C).

4.2. *Résultats*

La seule spécification que l'on fournit à NuSMV est la propriété :

ϕ : L´état Erreur de l'automate vérificateur est-il accessible ?

Initiateur	Temps processeur	Mémoire	Variables	Noeuds	Conformité
I1	6,9s	5Mo	217	12000	oui
I2	9,8s	5Mo	217	10500	non
I3	6,4s	5Mo	215	9500	oui
I4	7,6s	5Mo	215	10600	non

Figure 10. *Résultats*

Le tableau de la figure 10 récapitule les résultats obtenus sur un PC Pentium III à 933 MHz avec 256 Mo de mémoire, en autorisant NuSMV à faire des réordonnancements dynamiques des variables suivant l'heuristique par défaut (méthode “swift”). Nous avons, par ailleurs, abstrait la sortie $rdata$ de la cible puisqu'aucune contrainte ne lui est imposée. Notons que lorsque l'initiateur ne respecte pas la norme, NuSMV fournit un exemple d'exécution erronée, ce qui facilite le travail de correction.

5. Conclusion

Les résultats obtenus sont très positifs : les ressources nécessaires à la vérification des exemples simples d'initiateurs présentés dans cet article sont très réduites.

Nous allons entreprendre la vérification de l'implémentation de la norme sur un composant virtuel "complet" et déterminer dans quelle mesure le passage de la description du composant vers notre formalisme peut être rendu automatique. Nous pensons d'ailleurs que l'abstraction de l'interface d'un composant complet ne devrait pas conduire à des initiateurs beaucoup plus complexes que ceux présentés ici.

Nous devons également étendre nos résultats aux protocoles BVCI et AVCI et être capables d'analyser le comportement de cibles. (NuSMV permet, par exemple, de calculer le nombre maximum de cycles entre une requête de l'initiateur et son acquittement par la cible.)

Dans la problématique des systèmes sur puce, il serait également intéressant de vérifier et/ou d'analyser des composants virtuels complets afin que la réutilisation des composants soit totalement transparente. Nous pensons ainsi tenter l'analyse d'une norme de bus virtuel.

6. Bibliographie

[BAR 97] BARRET G., MCISAAC A., « Model Checking in a Microprocessor Design Project », *Proc. 9th International Conference on Computer Aided Verification (CAV'97)*, vol. 1254 de *Lecture Notes in Computer Science*, Springer, 1997, p. 214–225.

[MCM 93] MCMILLAN K. L., *Symbolic Model Checking*, Kluwer Academic Publishers, 1993.

[SCH 01] SCHNOEBELEN P., BÉRARD B., BIDOIT M., LAROUSSINIE F., PETIT A., *Model-Checking Techniques and Tools*, Springer, 2001.

[TUR 97] TURK A. L., T. P. S., POWERS G. J., « Verification of a Chemical Process Leak Test Procedure », *Proc. 9th International Conference on Computer Aided Verification (CAV'97)*, vol. 1254 de *Lecture Notes in Computer Science*, Springer, 1997, p. 84–94.

[VSI 01] VSI ALLIANCE(TM), *Virtual Component Interface Standard, Version 2 (OCB 2 2.0)*, avril 2001.

Annexe

On précise ici la description de l'automate vérificateur.

L'état initial du vérificateur est défini par la formule : $\neg req.val \wedge req.eop \wedge req.rd \wedge (req.address = 0) \wedge (req.be = 15) \wedge (req.wdata = 0)$.

Afin de préciser les macros, on se donne tout d'abord un certain nombre de prédicats auxiliaires :

$address\_plus\_quatre := (address = (req.address + 4)\ mod\ 256)$

$nouvelle\_requete := \neg req.val \wedge val$

$requete\_satisfaite := val \wedge ack$

$enregistrement\_nouvelle\_requete := (req.eop' = eop) \wedge (req.rd' = rd) \wedge (req.address' = address) \wedge (req.be' = be) \wedge (req.wdata' = wdata)$

$maintien\_requete := (req.eop' = req.eop) \wedge (req.be' = req.be) \wedge (req.rd' = req.rd) \wedge (req.address' = req.address) \wedge (req.wdata' = req.wdata)$

Nous pouvons à présent préciser les deux macros principales ($Erreur$ et MaJ_V) :

$Erreur := \Big(req.val \wedge$ (* requête insatisfaite *)

$\big(\neg val$
$\vee (req.rd \neq rd)$
$\vee (req.address \neq address)$
$\vee (req.wdata \neq wdata)\big)\Big)$ (* non maintien des requêtes *)

$\vee$

$\Big(nouvelle\_requete \wedge$

$\big((\neg req.eop \wedge (req.rd \neq rd \vee \neg(address\_plus\_quatre)))$

(* une requête d'un paquet doit avoir la même direction que la précédente et une adresse consécutive*)

$\vee (\neg eop \wedge req.eop \wedge \neg(address = 0\ mod\ 4))$

(* la première requête d'un paquet doit avoir une adresse divisible par 4 *)

$\vee (be \notin \{0, 1, 2, 4, 8, 3, 12, 15\})\big)\Big)$

(* le champ be doit etre valide *)

$MaJ_V :=$
if $(nouvelle\_requete \wedge \neg ack)$
then $(enregistrement\_nouvelle\_requete \wedge req.val')$
(* En cas de nouvelle requête non satisfaite, on mémorise la requête et on active l'indicateur de requête insatisfaite*)
else if $(nouvelle\_requete \wedge ack)$
then $(enregistrement\_nouvelle\_requete \wedge \neg req.val')$
(* Sinon, en cas de nouvelle requête satisfaite, on mémorise la requête et on annule l'indicateur de requête insatisfaite*)

else if $(requete\_satisfaite)$ then $(\neg req.val' \wedge maintien\_requete)$
(* Sinon, si la requête en cours est satisfaite, on conserve la requête et on annule l'indicateur de requête insatisfaite*)
else $(maintien\_requete \wedge (req.val' = req.val))$
(* Dans les autres cas, on maintient les valeurs des variables req. $*$ *)