Vérification automatique de propriétés d'ordonnanceurs Bossa

J.-P. Bodeveix, M. Filali
IRIT
Université Paul Sabatier
118 route de Narbonne
F-31062 Toulouse cedex
France
Email:{bodeveix,filali}@irit.fr

J. Lawall
DIKU
University of Copenhagen
2100 Copenhagen
Denmark
Email:julia@diku.dk

G. Muller
Ecole des Mines de Nantes
INRIA, LINA
F-44307 Nantes cedex 3
France
Email:Gilles.Muller@emn.fr

Résumé

Bossa est un environnement dédié au développement d'ordonnanceurs. Dans cet article, nous étudions l'automatisation de la vérification des propriétés énoncées par le langage dédié de Bossa. Nous montrons que la plupart de ces propriétés peuvent être vues commes des propriétés d'invariance ou de raffinement. Pour automatiser les obligations de preuve associées à ces notions, nous utilisons la logique WS1S et l'outil Mona qui implante une procédure de décision de cette logique. Les techniques de preuve utilisées sont exprimées à l'aide de l'outil FMona.

1 Introduction

Bossa [11] est un environnement dédié au développement d'ordonnanceurs hiérarchiques. Il a été effectivement utilisé dans des systèmes tels que Linux ou Chorus [10]. Bossa offre un langage de spécification adapté à la description de l'ordonnanceur d'un système opératoire et un langage de programmation dédié à l'implantation de politiques d'ordonnancement de processus. Le langage de spécification est utilisé une fois par un spécialiste lors du portage de Bossa sur un nouveau système alors que le langage d'implantation doit être utilisable par un programmeur d'application ayant besoin d'une politique d'ordonnancement spécifique. Les deux langages offrent des abstractions adaptées aux problèmes d'ordonnancement, ce qui facilite notamment une description intuitive (métier) de ces problèmes, une génération de code efficace et une vérification naturelle de la compatibilité de la politique avec le système cible.

Bossa possède une sémantique formelle bien définie et un vérificateur ad hoc a été développé [11]. Les propriétés vérifiées portent soit sur la spécification des interactions noyau/ordonnanceur, soit sur l'implantation d'une politique donnée. Le langage de spécification décrit le séquencement des événements traités par l'ordonnanceur et les transitions autorisées lors du traitement d'un événement. La principale propriété vérifiée concerne l'absence de blocage du système. Le langage d'implantation offre des primitives facilitant le codage sûr de handlers d'événements. On vérifie principalement la conformité du code avec la spécification. La vérification de ces propriétés ne peut en général être réalisée localement et exploite la dynamique du système.

Le vérificateur inclus dans l'environnement Bossa effectue des analyses utilisant des techniques d'interprétation abstraite. De par sa spécificité, ce vérificateur est complexe et difficile à maintenir notamment lors des évolutions de Bossa. Dans un premier travail [4], nous avions étudié comment la méthodologie Bossa pouvait s'inscrire dans le cadre de la méthode formelle B [1]. Cette première étude a montré que certains contrôles effectués par Bossa pouvaient être exprimés comme des obligations de preuve associées par exemple à des invariants ou à des raffinements B. Cependant, la plupart de ces obligations de preuve n'étaient pas automatiquement «déchargées» par les prouveurs automatiques de l'atelier B [6]. Parmi les raisons, citons l'absence de procédures de décision et de générateur d'abstraction adaptés. Le travail présenté ici est une extension de notre travail précédent : d'une part, nous complétons la couverture des propriétés vérifiées par Bossa et d'autre part,

nous montrons comment la plupart de ces preuves peuvent être automatisées par des procédures de décision standards et des abstractions dédiées.

Dans le reste de cet article, après avoir introduit (section 2) les outils et techniques de preuve utilisés, nous présentons (section 3) la modélisation du comportement générique des ordonnanceurs Bossa. La section 4 décrit la vérification de propriétés sur les événements typés décrivant ce comportement. La section 5 décrit l'expression de politiques d'ordonnancement et la vérification de leur conformité avec les types d'événements. Enfin, dans la section 6, nous établissons des liens avec d'autres méthodes et techniques de preuve.

2 Vérification en Mona et FMona

Dans cette section, nous présentons tout d'abord la logique de base de l'outil Mona [9] dont la procédure de décision a permis de vérifier automatiquement les propriétés de correction énoncées dans Bossa. Nous présentons ensuite l'outil FMona qui nous a permis d'énoncer de manière générique les propriétés Bossa ainsi que les techniques de vérification mises en oeuvre.

D'un point de vue sémantique, nous considérons les systèmes de transitions qui constituent un formalisme de base pour modéliser les systèmes. Nous présentons l'expression FMona de ces systèmes, les propriétés de base que nous utiliserons pour la spécification et, enfin, les techniques de vérification associées.

2.1 L'outil de décision Mona pour la logique WS1S

Définition 1 (Les logiques S1S et WS1S) *Soit $\{x_1, \ldots, x_n\}$ un ensemble de variables du premier ordre et $\{X_1, \ldots, X_n\}$ un ensemble de variables du second ordre. Une grammaire minimale pour ces logiques est définie ainsi :*

– *un terme t est défini inductivement par :*

$$t \ ::= 0 \mid x_i \mid s(t) \text{ où s est le symbole successeur}$$

– *une formule f est définie inductivement par :*

$$\begin{array}{rl} f \ ::= & t \in X_i \ \text{appartenance} \\ & \mid \neg f \mid f \wedge f \\ & \mid \exists_1 x_i.\ f \ \text{quantification du premier ordre} \\ & \mid \exists_2 X_i.\ f \ \text{quantification du second ordre (sur des ensembles)} \end{array}$$

De manière classique cette syntaxe est étendue avec des opérateurs logiques et des quantificateurs ($\vee, \Rightarrow, \forall \ldots$) ainsi que des relations arithmétiques. A titre d'exemple : $a \leq b$ est défini par $\forall_2 X.\ a \in X \wedge (\forall_1 x.\ x \in X \Rightarrow s(x) \in X) \Rightarrow b \in X$.

Validité d'une formule. Une formule close est valide en S1S ou WS1S si son interprétation sur l'ensemble $\mathbb{N}$ des entiers naturels est valide, où s est la fonction successeur, les variables du premier ordre représentent des entiers naturels et les variables du second ordre des sous ensembles (finis dans le cas de WS1S) de $\mathbb{N}$. Ainsi, la formule $\forall_2 X : \exists_1 x : \neg x \in X$ est valide en WS1S mais pas en S1S. Ces deux logiques sont décidables. L'outil Mona [9] implante une procédure de décision pour WS1S.

2.2 FMona

L'outil FMona [3] est une interface de haut niveau pour Mona. Par exemple, il est possible de déclarer des types énumérés, des enregistrements avec mise à jour, et de quantifier sur des types finis composés. De plus, FMona permet la définition de macros d'ordre supérieur paramétrées par des types et des prédicats. Le code source FMona est vérifié du point de vue typage, macro expansé et converti en Mona.

Par exemple, on peut définir en FMona la notion de superposition [5]. Dans l'exemple suivant, on suppose d'une part que l'on a un automate dont les états sont identifiés par le type `Location`

et d'autre part une relation de transition `tr` sur des données de type `Data`. La superposition à une transition de l'automate de la relation `tr` nous donne une nouvelle relation de transition qui sera définie sur le type «superposé» : **record**`{d :Data; w :Location;}` ainsi :

```
# superposition de transitions de l'automate et de la mise à jour des données
pred superpose(type Data, type Location, pred(var Data d,d') tr, var Location l,l',
                var record{d: Data; w: Location;} s,s') =
  s.w = l ∧ s'.w = l' ∧ tr(s.d,s'.d);
```

Remarque : en FMona, il est possible d'appliquer partiellement un prédicat ; dans l'exemple précédent, soit une relation `tr` et deux locations `l1,l2`, alors `superpose(tr,l1,l2)` est un prédicat sur des couples de type superposé.

2.3 Systèmes de transitions

Un système de transitions est défini par un triplet comprenant un espace d'état, un prédicat d'initialisation et une relation binaire dite de transition. Par exemple, le code ci-dessous décrit l'espace d'état, le prédicat d'initialisation et une transition d'un système de gestion de processus.

```
var nat NProc; # taille du système
type Proc  = ... NProc;  # type intervalle 0..NProc-1
type State = record{running, ready, blocked: set of Proc;};
# état initial du système
pred Init(var State s, var Proc boot) =
  s.running = {boot} ∧ s.ready = ∅ ∧ s.blocked = ∅;
# transfert de controle de p1 vers p2
pred Transfer(var Proc p1,p2, var State s,s') =
  p1 ∈ s.running ∧ p2 ∈ s.ready ∧
  s' = s with {running := {p2}; ready := (s.ready \ {p2}) ∪ {p1};};
```

Remarque : ce système est paramétré par `NProc`, mais Mona peut analyser des propriétés sans que le paramètre `NProc` n'ait été instancié. Lorsque la propriété est invalide, Mona génère un contre exemple portant sur ses variables libres. Ici, le paramètre `NProc` étant libre dans toutes les propriétés considérées, un contre-exemple donnera une valeur pour laquelle la propriété n'est pas satisfaite.

2.4 Propriétés

Dans cet article, nous considérons des propriétés de sûreté exprimant qu'«un état indésirable n'est jamais atteint». Les propriétés de vivacité exprimant qu'«un état désiré finira par être atteint» ne sont donc pas abordées. En l'occurrence, nous considérons les propriétés d'invariance et de validité dans toute exécution. Pour être homogène avec la terminologie Mona, nous parlerons dans la suite de prédicat plutôt que de propriété.

Définition 2 (Invariant) *Un prédicat P est dit invariant dans un système de transitions s'il est vérifié par les états initiaux et préservé par toute transition (c'est-à-dire si P est stable [5]).*

En FMona, nous introduisons les macros `stable` et `invariant` prenant en paramètre le système de transitions défini par le type de son espace d'état, son prédicat d'initialisation et sa relation de transition, et le prédicat que l'on souhaite montrer stable ou invariant :

```
pred stable(type State, pred(var State s) inv, pred(var State s,s') tr) =
  all State s,s': inv(s) ⇒ tr(s,s') ⇒ inv(s');

pred invariant(type State,
  pred(var State s) inv,pred(var State s) init,pred(var State s,s') tr) =
  (all State s: init(s) ⇒ inv(s)) ∧ stable(inv,tr);
```

La propriété d'invariance repose sur la stabilité et ne tient donc pas compte des exécutions effectivement possibles. Lorsque l'invariance d'une propriété ne peut pas être établie, nous pouvons vérifier une propriété plus faible : est-elle *toujours vraie* ? C'est-à-dire satisfaite par tout état accessible de toute exécution.

Définition 3 (prédicat toujours vrai) *Un état s est dit accessible dans un système de transitions S si il existe une séquence $s_0, \ldots, s_n = s$ telle que s_0 est initial et pour tout $i < n$ il existe une transition de s_i à s_{i+1}. On note $\mathcal{A}cc(S)$ l'ensemble des états accessibles de S. Le prédicat P est dit toujours vrai sur S si P est vérifié par tout état accessible, ce qui est noté $S \models P$.*

En FMona, nous introduisons la macro `always_true` prenant en paramètre un système de transitions et le prédicat à vérifier.

```
pred reachable(type State, pred(var State s) init, pred(var State s,s') tr, var State s) =
  ex array nat of State A: ex nat i:
     A[i]=s ∧ init(A[0]) ∧ all nat j where j < i: tr(A[j],A[j+1]);

pred always_true(type State,
  pred(var State s) p,pred(var State s) init,pred(var State s,s') tr) =
  all State s: reachable(init,tr,s) ⇒ p(s);
```

2.5 Techniques de vérification

Dans cette section, nous exprimons à l'aide de FMona, les techniques de vérification que nous utilisons : le raffinement permettant d'établir la conformité d'une spécification concrète avec une spécification abstraite, l'abstraction permettant de vérifier une propriété sur un système en construisant un système abstrait plus petit, et une méthode itérative de recherche de point fixe.

2.5.1 Expression de la technique du raffinement

La notion de raffinement entre un système dit abstrait et un système dit concret exprime que tout comportement du système concret est aussi un comportement du système abstrait. Pour établir cette propriété, nous utilisons dans cette étude le raffinement avant [7].

Un système de transitions dit concret S_c raffine un système de transitions dit abstrait S_a si il existe une relation φ entre les données du modèle concret et celles du modèle abstrait, appelée invariant de liaison, telle que :

- à tout état initial du modèle concret correspond par φ un état initial du modèle abstrait.
- Etant donné un état concret `c` et un état abstrait `a` en relation par φ, à tout état `c'` image de `c` par la relation de transition concrète, il existe un état abstrait `a'`, image de `a` par la relation de transition abstraite, en relation par φ avec `c'`.

En FMona, nous introduisons les macros `refines_init` et `refines` exprimant les deux conditions précédentes :

```
pred refines_init(type State_c, type State_a,
                  pred(var State_c s_c,var State_a s_a) φ,
                  pred(var State_c s_c) init_c,
                  pred(var State_a s_a) init_a
                  ) =
 all State_c s_c: init_c(s_c) ⇒ ex State_a s_a: init_a(s_a) ∧ φ(s_c,s_a);

pred refines(type State_c, type State_a, pred(var State_c s_c,var State_a s_a) φ,
             pred(var State_c s_c, s_c') tr_c, pred(var State_a s_a, s_a') tr_a) =
  all State_c s_c,s_c': all State_a s_a:
    (tr_c(s_c,s_c') ∧ φ(s_c,s_a)) ⇒ ex State_a s_a': tr_a(s_a,s_a') ∧ φ(s_c',s_a')
```

Le domaine de définition d'une transition que nous appellerons précondition doit aussi être pris en compte pour exprimer le principe de substitutivité. Une transition abstraite définie en un point donné ne peut être raffinée que par une transition concrète définie en ce même point. Le domaine

de définition modélise ici la précondition de la transition : une transition ne doit être applicable que lorsque sa précondition est satisfaite. La macro suivante définit la notion de précondition :

```
pred pre(type T, type T', pred(var T s1, var T' s2) tr, var T s) =
  ex T' s': tr(s,s');
```

En considérant la notion de précondition, la définition du raffinement doit être revue conformément à la définition proposée par la méthode B : l'état abstrait doit satisfaire la précondition de la transition abstraite et toute concrétisation d'un état abstrait satisfaisant la précondition abstraite doit satisfaire la précondition de la transition concrète. L'obligation de preuve du raffinement devient alors la suivante :

```
pred refines_pre(type State_c, type State_a, pred(var State_c s_c,var State_a s_a) φ,
                 pred(var State_c s_c, s_c') tr_c , pred(var State_a s_a, s_a') tr_a) =
  (all State_c s_c,s_c': all State_a s_a:
    (tr_c(s_c,s_c') ∧ φ(s_c,s_a) ∧ pre(tr_a,s_a))
        ⇒ ex State_a s_a': tr_a(s_a,s_a') ∧ φ(s_c',s_a')) ∧
  all State_c s_c: all State_a s_a: (φ(s_c,s_a) ∧ pre(tr_a,s_a)) ⇒ pre(tr_c,s_c);
```

2.5.2 Expression de la technique d'abstraction

Tout d'abord rappelons que la technique d'abstraction consiste à définir un système (généralement) fini réduit à partir d'un système concret et d'une relation dite d'abstraction et à déduire des propriétés du système concret à partir de propriétés effectivement calculées sur le système abstrait. Il s'agit là d'une démarche duale à celle du raffinement dans le sens où, ici, notre point de départ est le système concret.

Etant donnés un espace d'états abstrait et une relation dite d'abstraction entre espaces concrets et abstraits, le système de transitions abstrait est introduit par la définition suivante :

Définition 4 (Abstraction) *Soit $C = (E_c, I_c, \rightarrow_c)$ un système de transitions, E_a un ensemble d'états dit «abstrait» et φ une relation sur $E_c \times E_a$. L'abstraction de C par φ est le système de transitions $(E_a, I_a, \rightarrow_a)$ où*

- *I_a est l'image par φ de I_c : $I_a = \varphi(I_c)$,*
- *$\rightarrow_a$ est l'ensemble des images par φ des paires connectées par $\rightarrow_c$.*

$$e_a \rightarrow_a e'_a \triangleq \exists e_c\, e'_c :\ \varphi(e_c, e_a) \wedge \varphi(e'_c, e'_a) \wedge e_c \rightarrow_c e'_c$$

FMona est utilisé pour synthétiser le système abstrait à partir de cette définition. Son expression est donnée par deux macros d'ordre supérieur définissant respectivement le prédicat d'initialisation et la relation de transition abstraits.

```
pred AInit(type State_c, type State_a
 ,pred(var State_c s_c) init_c
 ,pred(var State_c s_c, var State_a s_a) φ
 ,var State_a e_a) =
  ex State_c e_c: φ(e_c,e_a) ∧ init_c(e_c);

pred ANext(type State_c, type State_a
 ,pred(var State_c s_c, s_c') tr_c
 ,pred(var State_c s_c, var State_a s_a) φ
 ,var State_a e_a,e_a') =
  ex State_c e_c,e_c': φ(e_c,e_a) ∧ φ(e_c',e_a') ∧ tr_c(e_c,e_c');
```

Le résultat de base énonce que lorsque la relation d'abstraction est totale, un système de transitions raffine son abstraction. De plus, étant donnée une propriété P sur l'espace d'état concret, si son abstraction $\text{Abs}_\varphi(P) = \lambda a : \forall c : \varphi(c, a) \Rightarrow P(c)$ est toujours vraie dans le système abstrait, alors P est toujours vrai dans le système concret.

Il est à noter que les transitions du système abstrait sont quantifiées sur le domaine concret et ne sont pas nécessairement propositionnelles même si le domaine abstrait est fini. Par suite, l'existence d'une transition entre deux états abstraits n'est pas nécessairement décidable. Toutefois, elle

le devient si les transitions concrètes et la relation d'abstraction sont WS1S ; il est alors possible de synthétiser un système de transitions fini. La méthode à suivre pour vérifier une propriété P est alors la suivante :

1. Construire un système de transitions abstrait.
2. Construire l'abstraction de P : $\text{Abs}_{\varphi}(P)$.
3. Vérifier l'abstraction de P sur les états accessibles du système de transitions abstrait.

2.5.3 Expression de techniques itératives

La plupart des propriétés étudiées peuvent s'exprimer par des points fixes. Cependant, lorsque l'espace d'état des problèmes considérés est infini ou de grande taille, la construction du point fixe n'est pas garantie théoriquement ou pratiquement. Cette section a pour but de proposer des techniques, exprimées par des macros FMona, permettant de calculer le point fixe en un nombre fini d'itérations, indépendemment de la taille des problèmes *paramétrés* considérés. Comme il n'y a pas de résultat de décidabilité, l'utilisateur *doit* fournir une borne au nombre d'itérations effectuées par les macros. Nous présentons ici l'implantation en FMona de l'analyse arrière utilisée dans la section 5.

La macro `backward` prend en paramètre le nombre n d'itérations, le système de transitions défini par les prédicats `init` et `tr` et la propriété à vérifier, `inv`. Il effectue n itérations (à l'aide de la macro récursive `iterate`) de l'inverse de relation de transition `tr` à partir de la négation du prédicat `inv` puis vérifie (`check_bwd`) que l'on a bien calculé tous les états atteignables (le point fixe a bien été atteint) , mais qu'aucun état initial n'appartient à l'ensemble obtenu. On vérifie ainsi qu'il n'y a pas de chemin possible entre l'état initial et un état qui ne vérifie pas les propriétés requises.

```
pred iterate(type State,
     var nat N, pred(var State s) start, pred(var State s,s') tr, var State s) =
     if N = 0 then start(s)
     else ex State s': iterate(N−1,start,tr,s') ∧ (s' = s | tr(s',s)) endif;

pred check_bwd(type State,
     pred(var State s,s') tr,pred(var State s) bad,pred(var State s) init)=
     stable(bad,inverse(tr)) ∧ all State s: init(s) ⇒ ∼bad(s);

pred backward(type State, var nat N,
 pred(var State s,s') tr, pred(var State s) init, pred(var State s) inv) =
     check_bwd(tr,iterate(N,NOT(inv),inverse(tr)),init);
```

3 Comportement générique d'un ordonnanceur

Bossa offre un langage de spécification de politiques d'ordonnancement dont il vérifie certaines propriétés avant de générer un code exécutable. Nous avons retenu ici trois notions décrivant le comportement générique de tout ordonnanceur :

– les types d'événements,
– le contexte d'invocation des événements,
– les interruptions.

Il est à noter qu'il s'agit là d'informations ne pouvant être fournies que par un expert du domaine.

3.1 Les types d'événements

Afin de permettre la vérification de politiques d'ordonnancement, Bossa décrit les interactions entre le noyau et l'ordonnanceur. Ces interactions sont spécifiées à l'aide de classes d'états sensées abstraire les états de toute politique d'ordonnancement. Bossa distingue les classes suivantes :

– RUNNING : processus actif (au plus un pour un mono-processeur)

– READY : processus prêts à s'exécuter
– KERNEL_BLOCKED : processus bloqués du point de vue du noyau
– POLICY_BLOCKED : processus bloqués du point de vue de l'ordonnanceur
– KERNEL_POLICY_BLOCKED : processus bloqués pour le noyau et l'ordonnanceur
– TERMINATED : processus ayant terminé leur exécution
– NOWHERE : processus non pris en charge par l'ordonnanceur

Les échanges entre le noyau et l'ordonnanceur sont réalisés via des événements dont le traitement consiste à effectuer des changements d'états. Bossa décrit à la fois des changements entre états de différentes classes et au sein d'une même classe. Nous ne prenons en compte ici que les changements de classes. Les changements d'état dans une même classe seront donc non contrôlés.

Bossa introduit la notion de type d'événement pour décrire les transitions d'état permises. Par exemple l'événement `block.*` est décrit par les trois règles ci-dessous indiquant que le processus recevant l'événement peut être le processus actif ou un processus prêt ou bloqué du point de vue de la politique. Dans les deux derniers cas, il ne doit pas y avoir de processus actif. Les parties droites précisent dans chaque cas la nouvelle classe d'état du processus. Un processus ne doit pas recevoir cet événement s'il se trouve dans une classe non mentionnée. Des préconditions sur l'événement `block.*` sont donc ainsi mentionnées.

```
block.* : [tgt in RUNNING] -> [tgt in KERNEL_BLOCKED]
block.* : [[] = RUNNING, tgt in READY] -> [tgt in KERNEL_BLOCKED]
block.* : [[] = RUNNING, tgt in POLICY_BLOCKED] -> [tgt in KERNEL_POLICY_BLOCKED]
```

3.2 Le contexte d'invocation des événements et les interruptions

Afin d'assurer ces préconditions et plus généralement d'effectuer des vérifications prenant en compte les exécutions possibles, Bossa décrit sous la forme d'un automate les enchaînements d'événements pouvant être effectués par le système. Cet automate est fourni par un expert du noyau du système avec lequel les ordonnaceurs décrits en Bossa doivent s'interfacer.

L'automate décrivant les séquences d'événements possibles est décrit par la figure 1 où les états représentés par des carrés sont les sources des transitions non interruptibles.

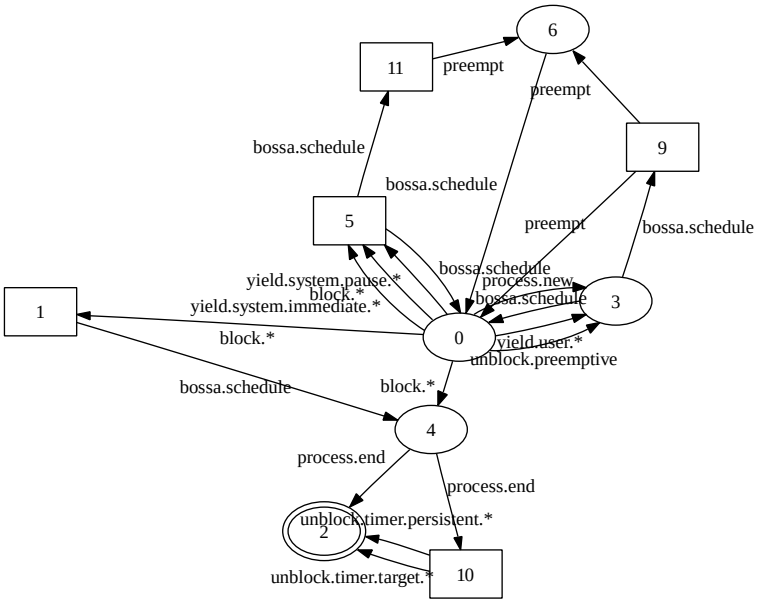

FIG. 1 – Automate d'appel

Les interruptions sont introduites au niveau des états spécifiés «interruptibles». Ces états, représentés par des cercles dans la figure 1, sont enrichis par des transitions réflexives étiquetées par les événements de type *interruption*.

Chaque chemin d'exécution de ce nouvel automate doit conduire à un état vérifiant les préconditions des événements obligatoires autorisés depuis cet état.

Par exemple, l'exécution ci-dessous est extraite de l'automate :

$$0 \xrightarrow{\text{yield.user.*}} 3 \xrightarrow{\text{bossa.schedule}} 9 \xrightarrow{\text{preempt}} 6 \xrightarrow{\text{bossa.schedule}} 0$$

Les types d'événements de Bossa déclarant les transitions issues de l'état 9 comme non interruptibles, les transitions d'interruption sont ajoutées aux autres états et dénotées ci-dessous par $\uparrow$:

$$0^{\uparrow^*} \xrightarrow{\text{yield.user.*}} 3^{\uparrow^*} \xrightarrow{\text{bossa.schedule}} 9 \xrightarrow{\text{preempt}} 6^{\uparrow^*} \xrightarrow{\text{bossa.schedule}} 0$$

4 Vérification de propriétés sur les types d'événements

Dans cette section, nous abordons la vérification au niveau des types d'événements. Après avoir présenté les principes de traduction, nous présentons les deux classes de propriétés vérifiées :

- les propriétés invariantes et plus généralement de conservation,
- la vérification du respect des préconditions.

L'expression FMona de ces propriétés assure leur décision automatique.

4.1 Traduction des types d'événements

Chaque classe est modélisée par l'ensemble des processus dont l'état fait partie de la classe indiquée. L'état du système est représenté par le type enregistrement `Classes` associant à chaque classe un ensemble de processus. Ce type ainsi que toutes les vérifications effectuées dépendent du nombre de processus présents dans le système, introduit par la variable globale `NProc`. Les vérifications sont réalisées indépendamment de ce paramètre.

```
var nat NProc;   #   nombre maximal de processus
type Proc = ... NProc;   #   type intervalle 0..NProc-1

type Classes = record {
  RUNNING: set of Proc;
  READY: set of Proc;
  POLICY_BLOCKED: set of Proc;
  KERNEL_BLOCKED: set of Proc;
  KERNEL_POLICY_BLOCKED: set of Proc;
  TERMINATED: set of Proc;
  NOWHERE: set of Proc;
};
```

A chaque déclaration de type d'un événement est associée une relation entre les états avant et après `s` et `s'`. Un prédicat, associé à chaque événement, est défini comme la disjonction des relations associées à chaque type. En supposant `block.*` défini par la seule déclaration ci-dessous :

```
block.* : [tgt in RUNNING] -> [tgt in KERNEL_BLOCKED]
```

les deux prédicats suivants sont automatiquement générés :

- `Eblock_` décrit une relation entre les états avant et après de type `Classes`, paramétrée par les processus `src` et `tgt` éventuellement référencés par la déclaration du type d'événement.

```
pred Eblock_(var Proc src, tgt, var Classes s,s') =
  (({tgt} sub s.RUNNING) ∧ (s' = s with {
    RUNNING := s.RUNNING \ {tgt};
    READY := s.READY \ {tgt};
    TERMINATED := s.TERMINATED \ {tgt};
    POLICY_BLOCKED := s.POLICY_BLOCKED \ {tgt};
    KERNEL_POLICY_BLOCKED := s.KERNEL_POLICY_BLOCKED \ {tgt};
    NOWHERE := s.NOWHERE \ {tgt};
    KERNEL_BLOCKED := s.KERNEL_BLOCKED \ {tgt} ∪ {tgt};
  }));
```

Pour des raisons de simplicité de génération de code, les processus mentionnés dans la partie gauche d'une règle sont systématiquement retirés de chaque classe. En effet, la partie gauche peut comporter une disjonction de cas qu'il est ainsi inutile d'examiner. Il est à noter que chercher à simplifier l'expression de la transition est inutile puisque `Mona` représente par un même automate toutes les formules équivalentes.

– `block_` définit l'union des transitions `Eblock_` pour chaque valeur possible des paramètres `src` et `tgt`.

```
pred block_(var Classes s,s') = ex Proc src, tgt: Eblock_(src,tgt,s,s');
```

4.2 Vérification de propriétés de conservation

Nous montrons ici comment exprimer et vérifier deux propriétés des types d'événements : la conservation de l'invariant de représentation des classes d'état et la conservation de l'ensemble des processus du système.

– La propriété `inv` exprime que les classes d'états sont disjointes deux à deux et que `RUNNING` contient au plus un processus. L'invariance de cette propriété est vérifiée pour chaque événement. Les prédicats suivants sont automatiquement générés puis vérifiés par FMona :

```
stable(inv,block_);
...
```

– La conservation du nombre de processus peut être vérifiée sur les types d'événements. On exprime pour cela que l'union des classes de processus est inchangée par le traitement d'un événement respectant sa spécification décrite par un type d'événement. Cette propriété est vérifiée sous l'hypothèse `inv` établie précédemment. On définit pour cela le prédicat `PasDePerte` prenant en argument une transition et exprimant que l'union des classes d'états est conservée par cette transition :

```
pred PasDePerte(pred(var Classes s,s') tr) =
  all Classes s,s': (inv(s) ∧ tr(s,s')) ⇒
    s.RUNNING ∪ s.READY ∪ s.POLICY_BLOCKED ∪ s.KERNEL_BLOCKED
    ∪ s.KERNEL_POLICY_BLOCKED ∪ s.TERMINATED ∪ s.NOWHERE =
    s'.RUNNING ∪ s'.READY ∪ s'.POLICY_BLOCKED ∪ s'.KERNEL_BLOCKED
    ∪ s'.KERNEL_POLICY_BLOCKED ∪ s'.TERMINATED ∪ s'.NOWHERE;
```

La propriété de conservation du nombre de processus est automatiquement générée pour chaque événement puis vérifiée par `FMona`.

```
PasDePerte(block_);
PasDePerte(detach);
PasDePerte(attach);
...
```

4.3 Vérification des préconditions

Les événements Bossa sont classés en événements obligatoires devant nécessairement être traités et événements optionnels pouvant être ignorés. Les événements possédant des préconditions, nous vérifions ici que les préconditions des événements obligatoires sont toujours satisfaites lorsque l'événement se produit. On considère pour cela l'automate décrivant l'ordonnancement des événements, complété par les événements d'interruptions.

A chaque état de l'automate est superposé l'état du système (décrit par le type `Classes`) associant à chaque classe l'ensemble des processus se trouvant dans cette classe. Aux transitions de l'automate se superposent celles décrites par les types d'événements.

En FMona, nous introduisons le type `NClasses` décrivant l'espace d'état superposé et de la macro générique `superpose`(2.2) :

```
type Location = ... 12; #   nombre d'états de l'automate

type NClasses = record {
  d: Classes;    #   associe à chaque classe un ensemble de processus
  w: Location;   #   numéro de l'état
```

```
};
```

La relation de transition est obtenue en prenant la disjonction des transitions associées aux événements et en ajoutant les interruptions aux états de l'automate d'appel (figure 1) les autorisant.

```
#   événements d'interruption
pred interrupts(var Classes s,s') =
    unblock_preemptive(s,s')
  | unblock_timer_target_(s,s')
  | unblock_timer_notarget_(s,s')
  | ...;

#   Etiquetage des transitions de l'automate par les événements
pred NNext(var NClasses s,s') =
    superpose(interrupts,2,2,s,s')
  | superpose(interrupts,4,4,s,s')
  | superpose(interrupts,6,6,s,s')
  | superpose(interrupts,0,0,s,s')
  | superpose(interrupts,3,3,s,s')
  | superpose(block_,0,1,s,s')
  | superpose(bossa_schedule,1,4,s,s')
  | ...
```

Il s'agit alors de montrer que tout état accessible de ce système satisfait les préconditions des événements obligatoires pouvant se produire depuis l'état correspondant dans l'automate. Cependant, déterminer les états accessibles de ce système de transitions n'est pas automatisable en général, l'espace d'états étant paramétré par le nombre de processus du système.

Il est donc nécessaire de procéder à une abstraction finie de l'espace d'état où à chaque classe est associé un booléen indiquant si la classe est vide (valeur `false`) ou non vide.

```
type AClasses = record {
  RUNNING: bool;
  READY: bool;
  POLICY_BLOCKED: bool;
  KERNEL_BLOCKED: bool;
  KERNEL_POLICY_BLOCKED: bool;
  TERMINATED: bool;
  NOWHERE: bool;
};
```

Ce type est complété par le numéro d'état de l'automate :

```
type NAClasses = record {
  d: AClasses;
  w: Location;
};
```

La relation d'abstraction entre l'espace des classes et l'espace défini par `AClasses` contient l'invariant de représentation `inv` et la définition de chaque champ booléen :

```
pred state_abs(var Classes s, var AClasses a) =
  inv(s) ∧
  (a.RUNNING ↔ s.RUNNING ≠ ∅) ∧
  (a.READY ↔ s.READY ≠ ∅) ∧
  (a.KERNEL_BLOCKED ↔ s.KERNEL_BLOCKED ≠ ∅) ∧
  (a.POLICY_BLOCKED ↔ s.POLICY_BLOCKED ≠ ∅) ∧
  (a.KERNEL_POLICY_BLOCKED ↔ s.KERNEL_POLICY_BLOCKED ≠ ∅) ∧
  (a.TERMINATED ↔ s.TERMINATED ≠ ∅) ∧
  (a.NOWHERE ↔ s.NOWHERE ≠ ∅);
```

La relation d'abstraction est étendue à l'automate complété par les transitions décrites à l'aide des types d'événements. On définit pour cela le prédicat `nabs` prenant en argument une donnée concrète de type `NClasses` et une donnée abstraite de type `NAClasses`. L'abstraction conserve les numéros d'état de l'automate et abstrait les ensembles de processus attachés à chaque classe en booléens.

```
pred nabs(var NClasses c, var NAClasses a) =
  c.w=a.w ∧ state_abs(c.d,a.d);
```

La propriété de respect des préconditions est énoncée au niveau abstrait : toute concrétisation d'un état abstrait doit vérifier les préconditions des événements obligatoires.

La propriété de satisfaction des préconditions s'énonce sur les états de type `NClasses`. Elle exprime que tous les événements obligatoires (c'est-à-dire autres que l'événement d'appel à l'ordonnanceur `bossa.schedule`) autorisés par l'automate à partir de l'état courant sont effectivement applicables.

```
pred check_pre(var NClasses s) =
   (s.w = 10 ⇒ pre(unblock_timer_target_,s.d)) ∧
   (s.w = 10 ⇒ pre(unblock_timer_persistent_,s.d)) ∧
   ...
```

Pour être vérifiable, la propriété `check_pre` est exprimée sur les états abstraits : toute concrétisation de l'état abstrait doit satisfaire les préconditions des événements obligatoires autorisés :

```
pred check(var NAClasses a) = all NClasses c: nabs(c,a) ⇒ check_pre(c);
```

Une analyse par chaînage arrière du système abstrait, convergeant ici en un pas, permet de vérifier cette propriété sur tous les états abstraits accessibles. Les macros `AInit` et `ANext` effectuent l'abstraction du prédicat d'initialisation et de la relation de transition du système concret. On lance sur le système abstrait la vérification du prédicat `check`.

```
backward(1, ANext(NNext,nabs), AInit(NInit,nabs),check);
```

Remarques

- Cette étude nous a permis de détecter une erreur dans l'abstraction du comportement du système. Cette erreur ne peut cependant pas se produire dans le système effectif. L'analyse effectuée par l'outil Bossa ne détecte pas cette erreur parce qu'il abstrait un ensemble de processus par un domaine à trois états (vide, nonvide, inconnu). Aucune vérification n'est faite sur un état inconnu. Le système ainsi obtenu est plus compact mais moins précis.
- Bien qu'il soit théoriquement possible de déterminer les états accessibles de l'automate abstrait (macro `reachable` en 2.4), la complexité du calcul nous a contraint à appliquer une méthode itérative. Sa convergence (ici en un pas) est cependant garantie puisque l'espace d'état est fini. L'utilisation d'un vérificateur de modèle, plus efficace que Mona pour analyser un système fini, n'aurait cependant pas été possible puisque la relation de transition sur le domaine `NAClasses` est paramétrée par le nombre de processus du système.

5 Vérification de politiques d'ordonnancement

Une politique d'ordonnancement est définie par la déclaration des différents états d'un processus et du traitement associé aux événements déclarés, appelé handler d'événement. Nous montrons ici comment sont spécifiés en Bossa et traduits en FMona les handlers d'événements puis comment vérifier d'une part la bonne formation des handlers, c'est-à-dire le respect du domaine de définition des expressions, et d'autre part la conformité des handlers avec les types d'événements.

5.1 Spécification des handlers d'événements

Les éléments de spécification d'une politique que nous prenons en compte ici sont la déclaration des états d'un processus et le traitement associé à un événement. Les états sont disjoints deux à deux et subdivisent les classes d'états introduites avec les types d'événements. Par exemple, la politique Rate Monotonic [12] déclare six états comme suit :

```
states = {
  RUNNING running : process;
  READY ready : select queue;
  READY yield : process;
  KERNEL_BLOCKED blocked : queue;
  POLICY_BLOCKED computation_ended : queue;
  TERMINATED terminated;
}
```

Ainsi, la classe READY se divise en deux états, `ready` et `yield`. A un état peut être associée une structure de données permettant de stocker les processus se trouvant dans cet état :

- La déclaration `process` indique qu'au plus un processus se trouve dans cet état. L'état correspondant, s'il est non vide, pourra être manipulé comme un processus.
- La déclaration `queue` indique une gestion FIFO des processus se trouvant dans cet état. Une file peut être déclarée de type `select queue`, ce qui permet d'en extraire un élément maximal pour une relation d'ordre donnée. Pour Rate Monotonic, la relation est déclarée en Bossa par :
  ```
  ordering_criteria = { lowest period }
  ```
- L'absence de déclaration (dans le cas de `terminated` ici) indique l'absence de stockage des processus dans cet état.

Ces informations sont automatiquement traduites en FMona sous la forme d'une déclaration de type associant à chaque état un ensemble de processus et d'un prédicat de liaison entre états et classes d'états. Ce prédicat permet de plus d'exprimer l'invariant de représentation des états en s'appuyant sur celui des classes d'états, défini par l'invariant `inv` :

```
type Bstates = record {
  running: set of Proc;
  ready: set of Proc;
  yield: set of Proc;
  blocked: set of Proc;
  computation_ended: set of Proc;
  terminated: set of Proc;
};
```

```
pred Bstates2Classes(var Bstates c, var Classes a) =
  inv(a)
  ∧ a.RUNNING = c.running
  ∧ a.READY = c.ready ∪ c.yield
  ∧ c.ready ∩ c.yield = ∅
  ∧ a.POLICY_BLOCKED = c.computation_ended
  ∧ a.KERNEL_BLOCKED = c.blocked
  ∧ a.TERMINATED = c.terminated
  ∧ a.KERNEL_POLICY_BLOCKED = ∅;
```

En Bossa, les handlers d'événements décrivent des transitions d'états pouvant dépendre de conditions portant sur des comparaisons de priorité ou des valeurs de variables d'états attachées aux processus. Les traitements réalisés peuvent être partiellement définis. Par exemple, l'accès en tant que processus à un état déclaré `process` n'est permis que si cet état est non vide.

La traduction en FMona des handlers d'événements est obtenue par calcul de plus faible précondition sur l'instruction Bossa. Une abstraction des portions de code non exprimables en WS1S (expressions arithmétiques, flottantes, ...) est aussi effectuée. Ne sont conservées que les instructions de mise à jour des files de processus et les tests portant sur ces files. La traduction incorpore certaines conditions de bonne formation des expressions Bossa, comme celles portant sur la présence ou l'absence de processus dans un état de type `process`.

Par exemple, le traitement associé à l'événement `block` est décrit par le code Bossa suivant consistant à changer l'état du processus destinataire en `blocked` :

```
On block.* {
  e.target => blocked;
}
```

La traduction en FMona de ce handler exprime la suppression du processus recevant l'événement de tous les états et son ajout à l'état `blocked`.

```
pred Bblock_(var Proc src, tgt, var Bstates s,s') =
 (s with {
  running := (s.running \ {tgt});
  ready := (s.ready \ {tgt});
```

```
  yield := (s.yield \ {tgt});
  blocked := (s.blocked ∪ {tgt});
  computation_ended := (s.computation_ended \ {tgt});
  terminated := (s.terminated \ {tgt});
  } = s')
```

Afin de garantir la bonne formation des handlers, des préconditions sont générées lorsque les constructions utilisées sont partiellement définies, comme par exemple, en présence d'une transition vers un état de cardinalité inférieure ou égale à un. Dans l'exemple ci-dessous, la transition vers `yield` n'est valide que si `yield` est vide.

```
On yield.system.pause.* {
  e.target => yield;
}
```

L'expression FMona de ce handler incorpore la condition `yield=`$\emptyset$ qui, lorsqu'elle n'est pas satisfaite, bloque la transition. Nous montrerons qu'il est possible de vérifier que ceci ne se produit pas.

```
pred Byield_system_pause_(var Proc src, tgt, var Bstates s,s') =
 ((s.yield = ∅) ∧
  (s with {
    running := (s.running \ {tgt});
    ready := (s.ready \ {tgt});
    yield := (s.yield ∪ {tgt});
    blocked := (s.blocked \ {tgt});
    computation_ended := (s.computation_ended \ {tgt});
    terminated := (s.terminated \ {tgt});
    } = s'));
```

5.2 Vérification d'une politique

Nous montrons ici comment exprimer en FMona la conformité d'une politique avec les types d'événements. Cette propriété est basée sur la notion de raffinement entre les transitions sur le type `Classes`, dites abstraites, et les transitions sur le type `Bstates`, dites concrètes. La propriété générique de raffinement pour des transitions préconditionnées est ici instanciée pour la problématique Bossa et exprimée sous la forme d'un prédicat portant sur l'espace concret :

```
pred check_ref(pred(var Proc s,t,var Bstates c,c') trc,
               pred(var Proc s,t,var Classes a,a') tra, var Bstates c) =
 all Bstates c': all Classes a: all Proc src,tgt:
  refines_pre(Bstates2Classes,trc,tra);
```

Cependant, cette propriété de raffinement n'est pas vérifiée en général. Par exemple, une transition vers l'état `yield` n'est possible que si cet état est vide. Si la classe associée, `READY`, n'est pas supposée vide dans la déclaration de type d'événement, il n'y a pas raffinement. Il est nécessaire de prendre en compte les exécutions permises par l'automate et de vérifier un raffinement conditionné par l'accessibilité d'une certaine abstraction du type `Bstates`. Dans cette abstraction, chaque état de la politique est représenté par un booléen indiquant s'il est vide ou non. La relation d'abstraction `Bstates2ABstates`, établit cette correspondance.

```
type ABStates = record {
  running: bool;
  ready: bool;
  yield: bool;
  blocked: bool;
  computation_ended: bool;
  terminated: bool;
};
```

```
pred Bstates2ABstates(var Bstates c, var ABstates a) =
  (a.running ↔ c.running ≠ ∅) ∧
  (a.ready ↔ c.ready ≠ ∅) ∧
  (a.yield ↔ c.yield ≠ ∅) ∧
  (a.blocked ↔ c.blocked ≠ ∅) ∧
  (a.computation_ended ↔ c.computation_ended ≠ ∅) ∧
  (a.terminated ↔ c.terminated ≠ ∅);
```

Aux états concrets et abstraits sont superposés ceux de l'automate. L'extension de la relation d'abstraction aux états superposés, `NBstates2NABstates`, permet de définir un système de transitions abstrait sur un espace d'état fini à partir du code Bossa. Nous vérifions alors la propriété de raffinement sur toutes les concrétisations possibles des états abstraits accessibles. Seules les transitions permises à partir d'un état donné de l'automate sont prises en compte :

```
type NABstates = record { d: ABstates; w: Location;};

pred check_aref(pred(var Proc s,t,var Bstates c,c') trc,
                pred(var Proc s,t,var Classes a,a') tra,
                var Location l, var NABstates a) =
  a.w=l ⇒ all Bstates c: Bstates2ABstates(c,a.d) ⇒ check_ref(trc,tra,c);
```

L'assertion ci-dessous exprime que l'on peut montrer en chaînage arrière que la propriété de raffinement pour l'événement `block.*` est satisfaite depuis tout état dont l'abstraction est accessible. La convergence est obtenue après trois applications arrières de transitions abstraites.

```
backward(3, ANext(NBNext,NBstates2NABstates), AInit(nBInit,NBstates2NABstates),
         check_aref(Bblock_,Eblock_,0));
```

Il est à noter que le nombre minimal d'itérations ne peut être déterminé sans lancer la vérification. Cette valeur pourrait être un paramètre du générateur de code FMona. Bien que les propriétés considérées soient paramétrées par le nombre de processus du système, la convergence de la méthode itérative est assurée par le fait que l'espace abstrait est fini. Le nombre maximal d'itérations est borné par la taille de cet espace (640 pour Rate Monotonic), mais trois itérations suffisent.

6 Travaux connexes

Dans cet article, nous avons étudié la vérification automatique de propriétés d'ordonnanceurs Bossa à l'aide de la logique décidable WS1S. D'autres méthodes et techniques de preuve auraient pu être utilisées.

En ce qui concerne les méthodes, citons la méthode B que nous avions utilisée dans notre étude précédente [4]. Hormis le fait, que dans cet article, nous avons explicitement établi que les obligations de preuve pouvaient s'exprimer dans une logique décidable et donc être déchargées automatiquement (moyennant la disponibilité d'une procédure de décision de cette logique), la prise en compte de l'automate de comportement Bossa n'est pas supportée naturellement par la méthode B. Plus précisément, la prise en compte de cet automate a nécessité d'expliciter une sur approximation du plus fort invariant synthétisée automatiquement à partir de la donnée d'une abstraction. Cette synthèse automatique qui correspondrait à une démarche d'interprétation abstraite n'est pas supportée par B qui préconise un développement inverse partant du point de vue le plus abstrait.

En ce qui concerne l'aspect technique de preuve, la modélisation des états et classes d'états par des compteurs indiquant le nombre de processus qu'ils contiennent aurait aussi été possible. Mona, supportant l'arithmétique de Presburger aurait donc aussi pu être utilisée, mais des outils comme FAST [2], dédiés à une telle arithmétique sont sans doute plus efficaces. La logique WS1S, nous a permis d'éviter l'étape d'abstraction des ensembles de processus par des compteurs. Enfin, nous aurions aussi pu modéliser ce problème à l'aide des réseaux de Petri [8], qui constituent un formalisme de modélisation où l'usage des compteurs est restreint mais qui devrait suffire pour cette modélisation. Dans les deux cas, les propriétés envisagées ne sont pas décidables automatiquement et nécessitent la mise en oeuvre d'abstractions ou l'application de techniques d'accélération de convergence.

7 Conclusion

Nous avons montré dans cet article comment spécifier dans une logique décidable la plupart des propriétés vérifiées par les outils spécifiques du langage Bossa. Ce travail peut être perçu comme un effort de spécification des algorithmes de vérification mis en oeuvre dans Bossa et permettrait aussi de séparer dans Bossa une activité de génération d'obligations de preuve et une activité de

vérification. Cette étude ne couvre pas toutes les analyses effectuées par Bossa et notamment l'aspect génération de code optimisé. Un couplage beaucoup plus fort entre Bossa et FMona serait nécessaire puisque des appels à la procédure de décision Mona devraient être insérés dans le générateur de code C de l'outil Bossa.

En terme de performances, l'utilisation d'un démonstrateur généraliste pour la logique WS1S est bien sûr pénalisante mais les temps d'exécution sont du même ordre de grandeur (de l'ordre de la minute) que ceux de l'outil dédié. L'utilisation de la technique itérative évitant l'utilisation de quantifications d'ordre deux améliore grandement les performances aussi bien en temps qu'en espace mémoire.

Les vérifications effectuées ici et par l'outil Bossa garantissent que le code de l'ordonnanceur s'intègre au noyau. Des propriétés liées à l'algorithme d'ordonnancement voire à l'application ordonnancée pourraient aussi être envisagées. Par exemple, la caractérisation d'un ordonnanceur particulier est-elle vérifiée par l'implantation ? L'application ordonnancée respecte-t-elle ses échéances ? Enfin, la génération de code certifié et la génération de code annoté par sa preuve [13] figurent aussi parmi les perspectives de cette étude.

Ce travail a été financé en partie par le Danish Research Council, subvention 21-03-0545 et en partie par le projet CORSS : *Composition et raffinement de systèmes sûrs* de l'ACI Sécurité Informatique du ministère de la recherche et des nouvelles technologies.

Références

[1] J.-R. Abrial. *The B-Book : Assigning programs to meanings*. Cambridge University Press, 1996.

[2] S. Bardin, A. Finkel, J. Leroux, and L. Petrucci. FAST : Fast acceleration of symbolic transition systems. In *Proc. 15th Conf. Computer Aided Verification (CAV'2003), Boulder,CO,USA*, volume 2725 of *Lecture Notes in Computer Science*, pages 118–121. Springer, 2003.

[3] J.-P. Bodeveix and M. Filali. A generic tool for expressing the development of validations . In *11th Nordic Workshop on Programming Theory NWPT'99, Uppsala*, pages 37–37, Uppsala University, 6-8 octobre 1999. Bjorn Victor and Wang Yi.

[4] J.-P. Bodeveix, M. Filali, J. Lawall, and G. Muller. Formal methods meet domain specific languages. In *Fifth International Conference on Integrated Formal Methods (IFM), Eindhoven Netherlands*, page (to appear). LNCS (springer verlag), 29 november-2 december 2005.

[5] K. Chandy and J. Misra. *Parallel Program Design, A Foundation*. Addison-Wesley, 1988.

[6] CLEARSY. L'atelier B. version 3.6. Technical report, http ://www.atelierb.societe.com/, 2005.

[7] W.-P. de Roever and K. Engelhardt. *Data Refinement Model-Oriented Proof methods and their Comparison*. Cambridge University Press, 1998.

[8] M. Diaz. *Vérification et mise en oeuvre des réseaux de Petri*. HERMES, 2003.

[9] J. Henriksen, J. Jensen, M. Jorgensen, N. Klarlund, R. Paige, T. Rauhe, and A. Sandholm. Mona : Monadic second-order logic in practice. In *Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, http ://www.brics.dk/∼mona, pages 58–73, Aarhus, May 1995.

[10] Jaluna. Jaluna Osware. http ://www.jaluna.com.

[11] J. Lawall, A.-F. Le Meur, and G. Muller. On designing a target-independent DSL for safe OS process-scheduling components. In *Third International Conference on Generative Programming and Component Engineering (GPCE'04)*, volume 3286 of *Lecture Notes in Computer Science*, pages 436–455, Vancouver, October 2004. Springer-Verlag.

[12] C. L. Liu and J. W. Layland. Scheduling algorithms for multi-programming in a hard real-time environment. *Journal of the Association for Computing Machinery*, 20(1) :40–61, january 1973.

[13] G. Necula and P. Lee. Safe kernel extensions without run-time checking. In *OSDI96*, pages 1–13, Seattle, october 1996.