Spécification du langage CDL version 2.1 : Syntaxe et sémantique 21 novembre 14

Philippe Dhaussy

Lab-Sticc, ENSTA Bretagne.
2, rue François Verny. 29806 Brest France
philippe.dhaussy@ensta-bretagne.fr

Résumé

Ce document décrit la syntaxe et la sémantique du langage CDL V.2.1 (Context Description Language).

Ce langage est exploité par l'outil OBP (Observer-Based Prover) disponible sur le site http ://www.obpcdl.org.

Table des matières

1 Introduction au langage CDL

Un modèle CDL permet à l'utilisateur de décrire le comportement de l'environnement du modèle à valider et les propriétés devant être vérifiées. Le comportement est considéré comme des enchainements de scénarios qui décrivent les interactions entre le modèle soumis à validation et des entités composant l'environnement de ce modèle.

Ce DSL permet, d'une part, de décrire le comportement de plusieurs entités (nommés *acteurs*) contribuant à l'environnement et pouvant s'exécuter en parallèle. D'autre part, il intègre un langage de description de propriétés reposant sur la notion de patron et des automates observateurs.

Le langage est compilé par l'outil OBP (Observer-Based Prover) (figure 1) pour générer des graphes de comportements exploitables soit par l'outil OBP Explorer, soit par l'outil TINA SELT [?] via le langage Fiacre [?].

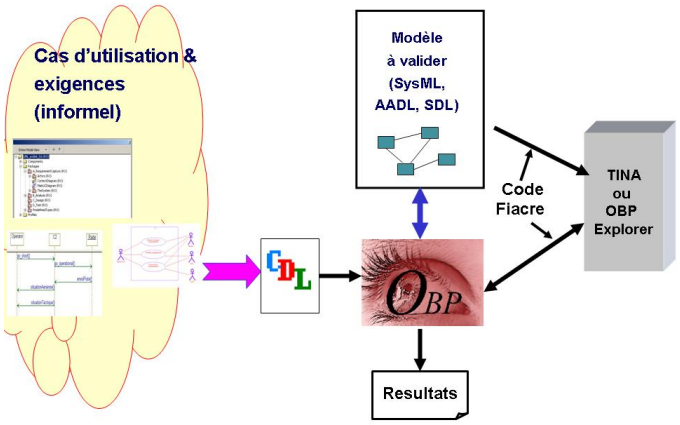

Figure 1. Transformation d'un modèle CDL avec OBP.

Lors de la compilation d'un modèle CDL, les comportements correspondant à chaque acteur sont dépliés (mis à plat) puis entrelacés pour respecter la sémantique du comportement parallèle des acteurs. Nous pouvons donc considérer que le modèle est structuré par un ensemble de diagrammes de séquences (type MSCs), reliés entre eux à l'aide de trois opérateurs : celui de la séquence (*seq*), l'opérateur parallèle (*par*) et l'alternative (*alt*). L'entrelacement d'un contexte, décrit par un ensemble de MSCs, génère un graphe représentant toutes les exécutions des acteurs de l'environnement considéré. Ce graphe est ensuite partitionné, de manière à générer un ensemble de sous-graphes correspondant aux sous-contextes, comme mentionné dans [?].

Lors de l'exploration par le vérificateur, chaque sous-graphe est composé avec le modèle à valider et les propriétés sont vérifiées sur le résultat de cette composition.

La sémantique de CDL est décrite section 3.

Pour une description des objectifs et concepts supportés par le langage et pour la présentation de l'outil OBP, voir les articles publiés [?,?,?,?] disponibles sur http ://www.obpcdl.org.

1.1 Exemple introductif

Nous illustrons le langage sur un petit exemple (figure 2). Un modèle de système comporte un processus SM. L'environnement est composé de 2 acteurs $Dev1$ et $Dev2$ qui interagissent avec le processus SM via un canal de communication asynchrone.

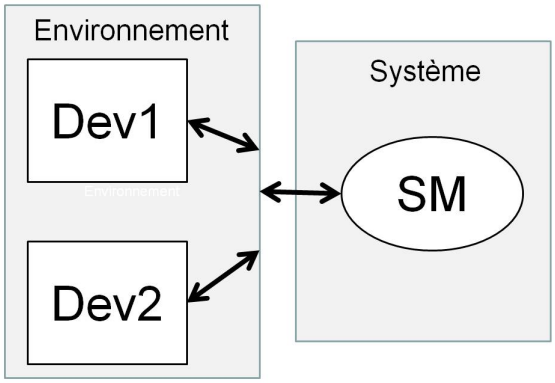

Figure 2. Exemple de système avec un environnement composé de 2 acteurs Dev1 et Dev2.

Un exemple de programme CDL est donné ici.

```
//--------- Event declarations ---------

event send_login1 is      {send login1 to {SM}1}
event recv_ack_log1 is    {receive ack_log1 from {SM}1}
event recv_nack_log1 is   {receive nack_log1 from {SM}1}
event send_operate1 is    {send operate1 to {SM}1}
event recv_ack_oper1 is   {receive ack_oper1 from {SM}1}
event recv_nack_oper1 is  {receive nack_oper1 from {SM}1}
event send_msg_end1 is    {send msg_end1 to {SM}1}

event send_login2 is      {send login2 to {SM}1}
event recv_ack_log2 is    {receive ack_log2 from {SM}1}
event recv_nack_log2 is   {receive nack_log2 from {SM}1}
event send_operate2 is    {send operate2 to {SM}1}
event recv_ack_oper2 is   {receive ack_oper2 from {SM}1}
event recv_nack_oper2 is  {receive nack_oper2 from {SM}1}
event send_msg_end2 is    {send msg_end2 to {SM}1}

//------- Predicat declarations -------

predicate SM_end is       {{SM}1@End }
predicate SM_var_0 is     {{SM}1:var = 0 }
```

```
//------- Event declarations with predicats -------

event evt_sm_end is { SM_end becomes true}
event evt_var_1 is  { SM_var_0 becomes false}


//---------  Property declarations ---------

//--- login1 is followed by ack_oper1 before SM reaches End state ---
property pte_login_ack is {
start -- / / send_login1 / -> wait;
wait  -- / / recv_ack_oper1 / -> start;
wait  -- / / recv_nack_oper1 / -> reject;
wait  -- / / evt_sm_end / -> reject
}

//--- End state of SM is reached -----
property pte_end is {
start -- / / evt_sm_end / -> success
}

//---------   Activity declarations ---------------

//---------- Dev1 <---> SM -----------
activity login1 is
        { event send_login1; { event recv_ack_log1 [] event recv_nack_log1 } }
activity operate1 is
        { event send_operate1; { event recv_ack_oper1 [] event recv_nack_oper1 } }
activity end1 is { event send_msg_end1 }

//---------- Dev2 <---> SM ------------
activity login2 is
        { event send_login2; { event recv_ack_log2 [] event recv_nack_log2 } }
activity operate2 is
        { event send_operate2; { event recv_ack_oper2 [] event recv_nack_oper2 } }
activity end2 is { event send_msg_end2 }

//---------- Actors : Environment behaviors ------------

activity DEV1 is
{
login1;
operate1;
end1
}
```

```
activity DEV2 is
{
login2;
operate2;
end2
}

//------------ CDL Scenarii -----------

cdl exemple_cdl is
{
properties pte_login_ack
assert SM_end

main is { DEV1 || DEV2 }
}
```

1.2 Commentaires de l'exemple

Déclarations des évènements.
Les évènements sont déclarés (mot clé *event*) en référençant des opérations d'émission *send...to* ou de réception *receive...from* de messages.

Exemple :

```
event send_login1 is      {send login1 to {SM}1}
event recv_ack_log1 is    {receive ack_log1 from {SM}1}
```

L'évènement *send_login*1 spécifie l'envoi, par l'environnement, du message *login*1 vers l'instance 1 du processus SM. L'évènement *recv_ack_log*1 spécifie la réception, par l'environnement, du message *ack_log*1 provenant de l'instance 1 du processus SM.

Les mesages *login*1 et *ack_log*1 sont déclarés, comme des constantes, dans le programe Fiacre du système.

Exemple de déclaration de contantes :

```
const login1       : nat  is 1
const ack_log1     : nat  is 2
```

Déclarations des prédicats.
Un prédicat est déclaré (mot clé *predicate*) en référençant soit un état d'un processus, soit une valeur de variable. Par exemple, le prédicat SM_end référence l'état End de l'instance 1 du processus SM. Il est vrai lorsque SM est dans l'état End, faux sinon. Le prédicat SM_var_0

référence la variable *var* de l'instance 1 du processus *SM*. Il est vrai lorsque *var* égale 0, faux sinon.

```
predicate SM_end is          {{SM}1@End }
predicate SM_var_0 is        {{SM}1:var = 0 }
```

Un prédicat peut être la négation (*not*) d'un prédicat, la disjonction (*or*) ou la conjonction (*and*) de deux prédicats.

Un évènement peut être déclaré à partir d'un prédicat. Il spécifie si le prédicat devient vrai ou faux. Par exemple, l'évènement *evt_sm_end* spécifie que le prédicat *SM_end* passe de faux à vrai. L'évènement *evt_var_1* spécifie que le prédicat *SM_var_0* passe de vrai à faux.

```
event evt_sm_end is { SM_end becomes true}
event evt_var_1 is  { SM_var_0 becomes false}
```

Déclarations des propriétés.
Une propriété est déclarée suivant deux formats : un format basé sur les patrons de définition de propriétés, un format basé sur les automates observateurs.

Pour les exemples de patrons, voir section 2.7.

Dans l'exemple ci-dessus, la propriété *pte_login_ack* est spécifié comme un automate comportant des transitions entre états de l'automate observateur. Chaque transition a le format suivant :

```
source -- g / p / e / r -> cible;
```

avec :
- *source* : l'état source de la transition,
- *g* : une garde sur horloge de l'observateur,
- *p* : un prédicat,
- *e* : un évènement,
- *r* : un reset de l'horloge de l'observateur
- *cible* : l'état cible de la transition.

Par exemple, la transition de l'automate observateur *pte_login_ack* :

```
start -- / / send_login1 / -> wait;
```

spécifie une transition entre les états *start* et *wait*, déclenchée par l'évènement *send_login*1.

Un automate observateur peut avoir, comme état cible, un état d'erreur (*reject*) comme dans la transition suivante pour la propriété *pte_login_ack* :

```
        wait -- / / recv_nack_oper1 / -> reject;
```

Il peut avoir, comme état cible, un état de succès (*success*) comme dans la transition suivante pour la propriété *pte_end* :

```
start -- / / evt_sm_end / -> success
```

Déclarations des activités.
Une activité est déclarée (mot clé *activity*) en référençant des évènements. Une activité est un évènement (par exemple l'activité $end1$), une séquence d'évènements spécifiée avec l'opérateur ; ou une alternative d'évènements spécifiée avec l'opérateur [] (par exemple l'activité $login1$)).

Déclarations des comportement des acteurs de l'environnement.
Le comportement des acteurs est déclarée comme une activité (par exemple $DEV1$ et $DEV2$).

Déclarations des scénarii de contexte.
Un scénario de contexte est déclaré comme une composition parallèle, avec l'opérateur ||, de comportement d'acteurs (par exemple le scénario *exemple_cdl*).

Un programme CDL peut comporter plusieurs scénarii.

Un scénario peut référencer une ou des propriétés et une ou des invariants sur les prédicats. Par exemple, le scénario *exemple_cdl* ci-dessous référence la propriété *pte_login_ack* et l'invariant spécifié avec le prédicat *SM_end*. Lors de l'exploration du modèle du système composé avec un des scénarii, *OBPExplorer* vérifie les propriétés et les invariants.

```
cdl exemple_cdl is
{
        properties pte_login_ack
        assert SM_end

        main is { DEV1 || DEV2 }
}
```

2 Syntaxe détaillée du langage CDL

La syntaxe du langage CDL est détaillée ici et les éléments de grammaire sont donnés. Les **mots en gras** sont des mots clés du langage. Les éléments de la grammaire entre crochets **[ ]** sont optionnels.

2.1 Structure d'un programme CDL

Un modèle CDL est déclaré par un programme dont la structure est donnée par la grammaire suivante :

prog_cdl_list ::= *prog_cdl prog_cdl_list*
| *prog_cdl*
;

prog_cdl ::= *event_declaration*
| *activity_declaration*
| *predicate_declaration*
| *property_declaration*
| *cdl_declaration*
;

2.2 Commentaires

Les commentaires commencent par /* et se termine par */

ou débutent à chaque ligne par //.

Exemple :

```
/*  Ceci est un commentaire
*/
// ceci est
// un commentaire.
```

2.3 Déclaration des variables et des litéraux

identifier_list ::= *identifier_list* , *IDENTIFIER*
| *IDENTIFIER*
;

literal_list ::= *literal_list* , *literal*
| *literal*
;

literal_field_list ::= *literal_field_list* , *literal_field*
| *literal_field*
;

literal_field ::= *IDENTIFIER* = *literal*
;

union_value ::= (*literal*)
;

literal ::= *INTEGER_LITERAL*
| *STRING_LITERAL*
| **true**
| **false**
| **any**
| *IDENTIFIER*
| *IDENTIFIER* [*literal_list*]
| *IDENTIFIER* { *literal_field_list* }
| *IDENTIFIER* . *IDENTIFIER* [*union_value*]
;

pid ::= { *IDENTIFIER* } *INTEGER_LITERAL*
| **any**
;

2.4 Déclaration des évènements

Les évènements servent à spécifier, lors de la construction des scénarii, des échanges entre l'environnement et les processus du sytème. Cet ensemble d'évènements sont détectables lors de l'exploration du modèle analysé et sont référencés dans les propriétés.

Les évènements sont :

- Emission asynchrone d'un message en provenance de l'environnement vers un processus
- Emission asynchrone d'un message en provenance d'un processus vers un processus
- Réception asynchrone d'un message en provenance d'un processus vers l'environnement
- Réception asynchrone d'un message en provenance d'un processus vers un processus
- Communication synchrone entre 2 processus
- Passage à *true* d'un prédicat
- Passage à *false* d'un prédicat

– Notification (*informal*)

Un message référencé dans un évènement est une valeur primitive (entière, booléenne) ou structurée (instanciée avec une structure de type *record*).

Les évènements d'émission et de réception sont utilisés pour la construction des activités composants les scénarios CDL (voir section 2.10).

Syntaxe

event_declaration ::= **event** *IDENTIFIER* **is** *braced_event*
;

braced_event ::= { [*options*] *event* }
;

event ::= **send** *message* **to** *pid* /* *par defaut* : *from ENV* */
| **send** *message* **from** *pid* **to** *pid*
| **receive** *message* **from** *pid* /* *par defaut* : *to ENV* */
| **receive** *message* **from** *pid* **to** *pid*
| **sync** *message* **from** *pid* **to** *pid*
| **informal** *#string#* **from** *pid*
| *predicate* **becomes true**
| *predicate* **becomes false**
;

message ::= *literal*
| *IDENTIFIER* ([*literal_list*])
;

Exemples d'évènements

Emission asynchrone :

Evènement e : Emission asynchrone du message m en provenance de l'environnement vers l'instance 1 du processus p :

```
event e is      {send m to {p}1}
event e is      {send m from {env}1 to {p}1}
```

Evènement e : Emission asynchrone du message m en provenance de l'instance 1 du processus p vers l'instance 1 du processus q :

```
event e is      {send m from {p}1 to {q}1}
```

Réception asynchrone :

Evènement e : Réception asynchrone d'un message en provenance de l'instance 1 du processus p vers l'environnement :

```
event e is    {receive m from {p}1}
event e is    {receive m from {p}1 to {env}1}
```

Evènement e : Réception asynchrone d'un message en provenance de l'environnement vers l'instance 1 du processus p :

(Mettre obligatoirement to p1)

```
event e is     {receive m from {env}1 to {p}1}
```

Evènement e : Réception asynchrone d'un message en provenance de l'instance 1 du processus p vers l'instance 1 du processus q :

```
event e is     {receive m from {p}1 to {q}1}
```

Communication synchrone :

Evènement e : Communication synchrone d'un message en provenance de l'instance 1 du processus p vers l'instance 1 du processus q via le port *po* (déclaré dans l'instance 1 du *component* *Comp*). Les différentes formes de déclarations sont :

```
event e is {sync {Comp}1:po (m) from {p}1 to {q}1}
event e is {sync {Comp}1:po () from {p}1 to {q}1}
event e is {sync {Comp}1:po (any) from {p}1 to {q}1}
event e is {sync {Comp}1:po (m) from any to {q}1}
event e is {sync {Comp}1:po (m) from {p}1 to any}
event e is {sync {Comp}1:po (m) from any to any}
```

Le mots clé *any* est utilisé pour désigner n'importe quel processus ou n'importe quelle valeur transmise. Pour les types d'événements précédents, les messages m sont déclarés comme constantes dans le programme Fiacre.

sync $\{Comp\}1 : po()$ si pas de paramètre dans po.

Si la valeur transmise est de type *record* (type T), on peut expliciter la valeur des champs dans la structure :

```
event e is {sync {Comp}1:po (T {v_1 = 2, v_2 = 3}) from {p}1 to {q}1}
event e is {sync {Comp}1:po (T {v_1 = 2, v_2 = any}) from {p}1 to {q}1}
```

Si la valeur transmise est de type *union* (type T), on peut expliciter la valeur de l'élément (par exemple $ELEMENT$) dans l'union :

```
event e is {sync {Comp}1:po (T.ELEMENT) from {p}1 to {q}1}
event e is {sync {Comp}1:po (T.ELEMENT) from {p}1 to {q}1}
```

Un évènement peut être déclaré à partir d'un prédicat. Il spécifie si le prédicat devient vrai ou faux.

Evènement e : Passage à *true* du prédicat *pred* :

```
event e is     {pred      becomes true}
```

Evènement e : Passage à $false$ du prédicat $pred$:

```
event e is    {pred      becomes false}
```

Evènement e : Notification de l'informal tag provenant d'un processus p :

```
event e is    {informal #tag# from {p}1 }
```

L'évènement informal peut être utilisé de plusieurs manière :

- Dans un programme Fiacre, l'informal est associé à un commentaire de type /*@tag*/ placé dans une transition d'un processus Fiacre. L'occurence de /*@tag*/ donne lieu à l'occurence de l'événement e spécifié pour marquer l'exécution dans une transition Fiacre.
- Dans le modèle CDL, l'événement e est utilisable pour marquer le passage dans une exécution CDL.
  Exemple :
  Une activity a est spécifiée comme la séquence de l'activity $a1$ suivie de e.

```
event e is    {informal #tag# from {p}1 }
activity a is  {a1; event e}
```

e marque le flot d'exécution dans l'activity a.

2.5 Déclaration des prédicats

Un prédicat est une expression booléenne référençant soit l'état d'un processus, soit la valeur d'une variable.

Syntaxe

predicate_declaration ::= **predicate** *IDENTIFIER* **is** { [*options*] *predicate* }
;

predicate ::= *predicate* **and** *not_predicate*
| *predicate* **or** *not_predicate*
| *not_predicate*
;

not_predicate ::= **not** *atomic_predicate*
| *atomic_predicate*
;

atomic_predicate ::= *pid* @ *IDENTIFIER*
| *comparison_predicate*
| (*predicate*)
| *IDENTIFIER*
| **all**
;

comparison_predicate ::= *system_access comparison_operator system_access*
| *system_access comparison_operator literal*
| *system_access* = *literal*
| *system_access* = *system_access*
;

comparison_operator ::= <
| <=
| >
| >=
;

system_access ::= *pid* : *system_fragment_list*
;

system_fragment_list ::= *system_fragment_list* . *system_fragment*
| *system_fragment*
;

system_fragment ::= *IDENTIFIER*
| *IDENTIFIER* [*INTEGER_LITERAL*]
;

Exemple de prédicats

Prédicat *pred* : l'instance 1 du processus *p* est dans l'état *s* :

```
predicate pred is    { {p}1@s}
```

Prédicat *pred* : la variable v de l'instance 1 du processus p est égale à la valeur n (n : entier ou true ou false) :

```
predicate pred is     { {p}1:v = n}
```

Prédicat *pred* : le troisième élément du tableau t de l'instance 1 du processus p est égale à la valeur n (n : entier ou true ou false) :

```
predicate pred      is { {p}1:t[2] = n }
```

Prédicat *pred* : la taille de la fifo f de l'instance 1 du composant c est égale à la valeur n (n : entier) :

```
predicate pred is     { {c}1:f.length = n}
```

Prédicat *pred*2 : négation du prédicat *pred*1

```
predicate pred2 is     { not pred1}
```

Prédicat *pred* : disjonction des prédicats *pred*1 et *pred*2

```
predicate pred is     { pred1 or pred2}
```

Prédicat *pred* : conjonction des prédicats *pred*1 et *pred*2

```
predicate pred is     { pred1 and pred2}
```

Prédicat *pred* : négation de la conjonction des prédicats *pred*1 et *pred*2

```
predicate pred is     { not (pred1 and pred2) }
```

Prédicat *pred* : conjonction et disjonction des prédicats *pred*1, *pred*2 et *pred*3

```
predicate pred is     { pred1 and (pred2 or pred3)}
```

2.6 Déclaration des propriétés

Deux types de déclaration des propriétés sont disponibles :
- Un type basée sur des patrons (textuel) de définition ;
- Un autre basée sur le format d'automates.

La syntaxe des propriétés est la suivante :

property_declaration ::= **property** *IDENTIFIER* **is** { [*options*] *parameterized_property* }
;

parameterized_property ::= *scope* *property*
;

property ::= *property_existence*
| *property_absence*
| *property_response*
| *property_observer*
;

scope ::= **strong scope** *predicate*
| **weak scope** *predicate*
| /∗ *nothing* ∗/
;

2.7 Déclaration des propriétés basées sur les patrons

La première forme de déclaration des propriétés est basée sur des patrons de définition : *Response*, *Existence* et *Absence*.

Les propriétés font référence à des évènements détectables (cf paragraphe 2.4).

L'objectif des propriétés pour les trois types prévus est le suivant :

- type *Response* : Une ou plusieurs occurrences des événements répond à une ou plusieurs occurrences des évènements.
- type *Absence* : Une ou plusieurs occurrences des événements ne doivent pas arriver durant l'exécution.
- type *Existence* : Une ou plusieurs occurrences des événements doivent arriver durant l'exécution avant un certain délai ou avant la fin chaque scénario.

Une propriété peut être rattachée à un contexte d'exécution (un scénario, voir en 2.11).

Syntaxe des patrons de propriétés

property_declaration ::= **property** *IDENTIFIER* **is** { [*options*] *parameterized_property* }
;

parameterized_property ::= *scope property*
;

property ::= *property_existence*
| *property_absence*
| *property_response*
| *property_observer*
;

property_existence ::= *occurrence_expression* **occurs** [*time_interval*] *repeatability*
;

property_absence ::= *occurrence_expression* **occurs never** [*time_interval*] *repeatability*
;

property_response ::= *occurrence_expression*
immediacy **leads − to** *time_interval*
occurrence_expression
nullity_list precedency_list
repeatability
;

property_precedence ::= *occurrence_expression*
immediacy **precedes** *time_interval*
occurrence_expression
nullity_list precedencyist
repeatability
;

immediacy ::= **immediately**
| **eventually**
;

occurrence_expression ::= *occurrences_description* { *occurrence_list* }
;

occurrences_description ::= **an**
| **all ordered**
| **all combined**
;

occurrence_list ::= *occurrence_list* ; *occurrence*
| *occurrence*
;

occurrence ::= *arity IDENTIFIER*
;

arity ::= **exactly one occurrence of**
| **one or more occurrences of**
.

Exemple de propriétés

Propriétés type Response

Type AN AN :

```
Property pty_an_an is {
        an {
        exactly one occurrence of e1;
        one or more occurrences of e2
           }
        eventually leads-to [0..5[
        an {
        exactly one occurrence of e3
           }
        one of e3 cannot occur before the first one of e1, e2
        repeatability : true
}
```

Type AN AN avec scope :

```
Property pty_an_an is {
  strong scope pred
        an {
        exactly one occurrence of e1;
        one or more occurrences of e2
           }
        eventually leads-to [0..5[
        an {
        exactly one occurrence of e3
           }
        one of e3 cannot occur before the first one of e1, e2
        repeatability : true
}
```

Dans la propriété précédente, *pred* est un prédicat. Si *pred* est faux avant l'occurence de $e1$, l'observateur n'est pas activé. Si *pred* est vrai avant l'occurence de $e1$, l'observateur est activé. Pour que l'observateur reste actif, il faut *pred* reste vrai.

On peut utiliser *weak* à la place de *strong*. Avec *weak*, si *pred* est faux avant l'occurence de $e1$, l'observateur n'est pas activé. Si *pred* est vrai avant l'occurence de $e1$, l'observateur est activé. L'observateur reste alors actif, même si *pred* devient faux.

Type ALL ALL :

```
Property pty_all_all is {
        all combined {
                    exactly one occurrence of e1;
                    exactly one occurrence of e2
```

```
        eventually leads-to [0..5[
        all ordered {
                    one or more occurrences of e3;
                    exactly one occurrence of e4;
                    exactly one occurrence of e5
        one of e3, e4, e5 cannot occur before the first one of e1, e2
        repeatability : true
}
```

Propriétés type Existence

```
Property existence_all_Combined is {
        all combined {
                exactly one occurrence of e1;
                exactly one occurrence of e2;
                exactly one occurrence of e3
        }
        occurs [1..10[
}
```

Propriétés type Absence

Type AN :

```
Property absence_an is {
        an {
                one or more occurrences of e1;
                exactly one occurrence of e2;
                one or more occurrences of e3
        }
        occurs never
```

Type ALL ORDER :

```
Property absence_all_order is {
        ALL ordered {
                exactly one occurrence of e1;
                exactly one occurrence of e2;
                exactly one occurrence of e3
        }
        occurs never
}
```

2.8 Déclaration des propriétés basées sur les automates observateurs

La deuxième forme de déclaration des propriétés est basée sur des automates observateurs. Ceux-ci font référence à des évènements détectables (cf section 2.4) et aux prédicats (cf section 2.5).

Comme pour le type de précédent de propriétés, basé sur les patrons, une propriété peut être rattachée à un contexte d'exécution (un scénario, voir en 2.11).

Syntaxe des automates d'observateurs

property_observer ::= *observer_transition_list*
| *clock* *observer_transition_list*
;

clock ::= **clock** *IDENTIFIER* ;
;

observer_transition_list ::= *observer_transition_list* ; *observer_transition*
| *observer_transition*
;

observer_transition ::= *IDENTIFIER* −−
transition_when /
transition_guard /
transition_event /
transition_reset
−> *IDENTIFIER*
;

transition_when ::= *IDENTIFIER* *comparison_operator* *INTEGER_LITERAL*
| /∗ *nothing* ∗/
;

transition_guard ::= *predicate*
| /∗ *nothing* ∗/
;

transition_event ::= *IDENTIFIER*
| /∗ *nothing* ∗/
;

transition_reset ::= *IDENTIFIER* := *INTEGER_LITERAL*
| /∗ *nothing* ∗/
;

Exemple de propriétés

Propriété non temporisée :

```
property pte_automata is {
        start -- / pred1 / e1 / -> s1;
        s1    -- / pred2 / e2 / -> start;
        s1    -- /       / e3 / -> s2;
        s2    -- /       / e4 / -> reject;
        s2    -- /       / e5 / -> success
}
```

Propriété non temporisée avec scope :

```
property pte_automata is {
  strong scope pred
        start -- / pred1 / e1 / -> s1;
        s1    -- / pred2 / e2 / -> start;
        s1    -- /       / e3 / -> s2;
        s2    -- /       / e4 / -> reject;
        s2    -- /       / e5 / -> success
}
```

Pour la sémantique du scope, voir l'exemple traité dans 2.7.

Propriété temporisée :

```
property pte_automata is {
  clock c;
        start --          / pred1 / e1 /          -> s1;
        s1    --          / pred2 / e2 / c:= 0    -> start;
        s1    -- c < 5    / pred3 / e3 /          -> s2;
        s2    -- c >= 5   /       / e4 / c:= -1   -> reject;
        s2    -- c < 5    /       / e5 /          -> success
}
```

c := -1 dans le champ *update* permet de dévalider l'horloge (cf règle *transition_reset*).

Note : si un champ d'une transition est vide, mettre au moins un espace entre les caractères '/' pour éviter que ce ne soit pris comme un commentaire "//".

2.9 Déclaration des invariants

Syntaxe

assert_list ::= *assert_list* ; *assert*
| *assert*
;

assert ::= **assert** *predicate*
;

Exemple de déclaration d'invariants

```
assert unPredicat;
assert {p}1@s;
assert {p}1:v = n
```

2.10 Déclaration des activités

Syntaxe des activités

activity_declaration ::= **activity** *IDENTIFIER* **is** *braced_activity*
;

braced_activity ::= **atomic** { *options activity* }
| **atomic** { *activity* }
| { *options activity* }
| { *activity* }
;

activity ::= *alt*
| *seq*
| *parameterized_activity*
;

alt ::= *alt* [] *parameterized_activity*
| *parameterized_activity* [] *parameterized_activity*
;

seq ::= *seq* ; *parameterized_activity*
| *parameterized_activity* ; *parameterized_activity*
;

loop ::= **loop** *INTEGER_LITERAL*;

parameterized_activity ::= *loop simple_activity*
| *simple_activity*
;

simple_activity ::= *IDENTIFIER* /* *Activities references* */
| *braced_activity* /* *Inlined activities* */
| *IDENTIFIER ASSIGN IDENTIFIER* /* *Assignment* */
| **event** *IDENTIFIER* /* *Events references* */
| **event** *braced_event* /* *Inlined events* */
| **skip** /* *Null Activity* */
;

Exemple d'activités

```
activity act1 is        { event e1 }
activity act2 is        { event e1; event e2  }
activity act3 is        { event e1 [] event e2  }
activity act3 is        { act1 [] act2  }
```

L'activité *act*4 spécifie une itération de 5 activités *act*3 :

```
activity act4 is        { loop 5 act3 }
```

L'activité *act*5 spécifie une séquence de 2 activités *act*1 et *act*2 exécutées de manière atomique :

```
activity act5 is atomic { act1, act2 }
```

2.11 Déclaration des scénarii

Syntaxe

cdl_declaration ::= **cdl** *IDENTIFIER* **is** {
[*options*]
[*properties*]
[*assert_list*]
[*init*]
main
}
;

properties ::= **properties** *identifier_list*
;

init ::= **init is** *parameterized_activity*
;

main ::= **main is** { *par*}
;

par ::= *par* || *parameterized_activity*
| *parameterized_activity*
;

Exemples de scénarii

Le scénario *scenar_1* met en œuvre 3 acteurs en parallèle dont les comportements sont respectivements *act*1, *act*2 et *act*3. Lors de l'exploration du modèle, les propriétés et invariants *pty*1, *pty*2, *pty*3, *inv*1 et *inv*2 sont testés.

```
cdl scenar_1 is
{
 properties pty1, pty2, pty3
 assert inv1;
 assert inv2;

main is { act1 || act2 || act3 }

}
```

Le scénario *scenar_2* met en œuvre 3 acteurs en parallèle avec une séquence d'initialisation comportant la séquence d'activités *act*1 et *act*2.

```
cdl scenar_2 is
{
  init is {
     act1;  act2
  }
        main is { act3 || act4 || act5 }
}
```

Le scénario *scenar_3* ne met en œuvre aucun acteur. Il sert uniquement à tester une propriété *pty*1 lor de l'exploration.

```
cdl scenar_3 is
{
 properties pty1

        main is { skip }
}
```

2.12 Déclaration des options

options ::= **options** { *option_list* }
| ;

option_list ::= *option_list* ; *option*
| *option* ;

option ::= *IDENTIFIER* = *STRING_LITERAL* ;

3 Sémantique formelle de CDL

3.1 Formalisation de CDL

Nous considérons qu'un contexte est décrit sous la forme d'une composition séquentielle, parallèle ou alternative, de diagrammes de séquences. Le langage CDL possède trois opérateurs simples à manipuler, *par*, *alt* et *seq*, respectivement notés par la suite $\|$, $+$ et ;, et qui permettent de structurer l'expression d'un scénario par leur combinaison. Pour des raisons de simplicité, nous ne considérons pas la notion des compteurs utilisée dans le langage et considérons des contextes simples sans boucle.

Formellement, un contexte est un système fini produisant et recevant des évènements décrits par la grammaire suivante :

$$\begin{aligned} C &::= M \mid C_1; C_2 \mid C_1 + C_2 \mid C_1 \| C_2 \\ M &::= \mathbf{0} \mid a!; M \mid a?; M \end{aligned}$$

Un contexte est soit un simple MSC M composé d'une séquence d'évènements d'émission $a!$ et de réception $a?$ terminée par un MSC terminal qui ne fait plus rien ($\mathbf{0}$), soit une composition séquentielle de deux contextes $(C_1; C_2)$, soit une alternative entre deux contextes $(C_1 + C_2)$, soit, enfin, une composition parallèle de deux contextes $(C_1 \| C_2)$.

3.2 Sémantique de CDL

Pour décrire la sémantique de CDL, considérons la fonction $wait(C)$ associant un contexte C avec l'ensemble des évènements attendus dans son état initial. Cette fonction est définie comme suit :

$$\begin{array}{lll} Wait\,(\mathbf{0}) \stackrel{\text{def}}{=} \emptyset & Wait\,(a!; M) \stackrel{\text{def}}{=} \emptyset & Wait\,(a?; M) \stackrel{\text{def}}{=} \{a\} \\ Wait\,(C_1 + C_2) \stackrel{\text{def}}{=} Wait\,(C_1) \cup Wait\,(C_2) & Wait\,(C_1; C_2) \stackrel{\text{def}}{=} Wait\,(C_1)\ si\ C_1 \neq \mathbf{0} & \\ Wait\,(\mathbf{0}; C_2) \stackrel{\text{def}}{=} Wait\,(C_2) & Wait\,(C_1 \| C_2) \stackrel{\text{def}}{=} Wait\,(C_1) \cup Wait\,(C_2) & \end{array}$$

Si d'un point de vue syntaxique, un contexte CDL peut être assimilé à un terme (sans récursion) d'une algèbre de processus telle que CCS ou LOTOS [**?**], en revanche la sémantique de CDL diffère de celle de ces algèbres de processus par le caractère asynchrone des communications entre un contexte CDL et le système auquel il est confronté. En effet, un contexte peut être assimilé à un processus communiquant de façon asynchrone avec le système via une file de messages mémorisant ses évènements d'entrée (reçus du système). De ce point de vue, la sémantique de CDL se rapproche de celle du langage SDL [**?**].

La sémantique du langage CDL est définie par une relation $(C, B) \xrightarrow{a} (C', B')$ pour exprimer que le contexte C, associé à un *buffer* B (une file d'attente d'évènements émis par le système à destination de son environnement), "produit l'action" a (qui peut être une émission ou une réception, voire l'événement $null_\sigma$ si C n'évolue pas), avant de devenir le nouveau contexte C' soumis à un nouveau *buffer* B'. Cette relation est définie par les règles suivantes (cf Figure 3) (dans l'ensemble de ces règles, a représente un événement différent de $null_\sigma$) :

- la règle *pref1* (sans pré-condition) spécifie qu'un MSC commence par une émission $a!$ puis poursuit le reste du MSC ;
- la règle *pref2* (sans pré-condition) spécifie qu'un MSC commence par une réception $a?$ et, confronté à un *buffer* d'entrée contenant en tête cet événement a, il consomme cet événement puis poursuit le reste du MSC ;

- la règle *seq1* spécifie qu'une séquence $C_1;C_2$ se comporte comme C_1 tant que celui-ci n'est pas terminé. La règle *seq2* spécifie en revanche que si C_1 s'achève (devient $\mathbf{0}$), la séquence devient C_2 ;
- les règles *par1* et *par2* spécifient que la sémantique de l'opérateur parallèle est basée sur une sémantique d'entrelacement asynchrone des exécutions ;
- la règle *alt* spécifie le choix entre deux contextes : $C_1 + C_2$ se comporte soit comme C_1, soit comme C_2. Mais une fois le choix fait, celui-ci est définitif (notons que cette règle a deux conclusions) ;
- enfin, la règle $discard_C$ spécifie que, si un événement a en tête du *buffer* d'entrée du contexte n'est pas attendu par celui-ci, alors il est perdu (supprimé de la tête du *buffer*).

$$\frac{}{(a!;M,B) \xrightarrow{a!} (M,B)}\ [\text{pref1}] \qquad \frac{}{(a?;M,a.B) \xrightarrow{a?} (M,B)}\ [\text{pref2}]$$

$$\frac{C_1' \neq \mathbf{0} \quad (C_1,B) \xrightarrow{a} (C_1',B')}{(C_1;C_2,B) \xrightarrow{a} (C_1';C_2,B')}\ [\text{seq1}] \qquad \frac{(C_1,B) \xrightarrow{a} (\mathbf{0},B')}{(C_1;C_2,B) \xrightarrow{a} (C_2,B')}\ [\text{seq2}]$$

$$\frac{C_1' \neq \mathbf{0} \quad (C_1,B) \xrightarrow{a} (C_1',B')}{\begin{array}{l}(C_1\|C_2,B) \xrightarrow{a} (C_1'\|C_2,B') \\ (C_2\|C_1,B) \xrightarrow{a} (C_2\|C_1',B')\end{array}}\ [\text{par1}] \qquad \frac{(C_1,B) \xrightarrow{a} (\mathbf{0},B')}{\begin{array}{l}(C_1\|C_2,B) \xrightarrow{a} (C_2,B') \\ (C_2\|C_1,B) \xrightarrow{a} (C_2,B')\end{array}}\ [\text{par2}]$$

$$\frac{(C_1,B) \xrightarrow{a} (C_1',B')}{\begin{array}{l}(C_1+C_2,B) \xrightarrow{a} (C_1',B') \\ (C_2+C_1,B) \xrightarrow{a} (C_1',B')\end{array}}\ [\text{alt}] \qquad \frac{a \notin wait(C)}{(C,a.B) \xrightarrow{null_\sigma} (C,B)}\ [\text{discard}_C]$$

Figure 3. Sémantique d'un contexte CDL

Notons que le caractère asynchrone de la composition parallèle (qui correspond à l'asynchronisme de l'environnement réel entourant le système) induit en pratique une génération importante du nombre des traces possibles d'un contexte. Par exemple, le contexte C suivant décrit 5040 traces différentes :

$C = (r1?;(s1! + s2!)) \parallel (s3!;r3?) \parallel (s4!;r4?) \parallel (s5!;r5?)$

3.3 Composition du contexte avec le système

Nous pouvons maintenant définir, comme une composition, la « fermeture » par un contexte $< (C,B_1) \mid (s,\mathcal{S},B_2) >$ d'un système $\mathcal{S}$ dans un état $s \in \Sigma$, Σ étant l'ensemble des états du système, avec son contexte C, B_2 et B_1 étant leurs buffers d'entrée respectifs (notons que chaque composant, système et contexte, a son propre buffer).

L'évolution de $\mathcal{S}$ fermé par C est donnée par deux relations : (1) la relation :

$$< (C, B_1)|(s, \mathcal{S}, B_2) > \overset{a}{\underset{\sigma}{\rightarrow}} < (C', B_1')|(s', \mathcal{S}, B_2') >$$

pour exprimer que $\mathcal{S}$ dans l'état s évolue vers l'état s' en recevant l'événement a, potentiellement vide ($null_e$), (produit par le contexte) et en produisant la séquence d'évènements σ, potentiellement vide ($null_\sigma$) (vers le contexte) ; et (2) la relation :

$$< (C, B_1)|(s, \mathcal{S}, B_2) > \overset{t}{\underset{\sigma}{\rightarrow}} < (C, B_1)|(s', \mathcal{S}, B_2') >$$

pour exprimer que $\mathcal{S}$ dans l'état s évolue vers l'état s' en laissant passer un temps t, et en produisant la séquence d'évènements σ, potentiellement vide ($null_\sigma$) (vers le contexte). Notons que dans le cas d'une évolution temporisée, seul le système évolue, le contexte n'étant pas temporisé.

La fermeture du système par son contexte est ainsi définie par les quatre règles suivantes : (cf Figure 4) :

- règle *cp1* : si $\mathcal{S}$ peut produire une séquence d'évènements σ, alors $\mathcal{S}$ évolue et σ est placé en fin du *buffer* de C ;
- règle *cp2* : si C peut émettre a, C évolue et a est placé en fin du *buffer* de $\mathcal{S}$;
- règle *cp3* : si C peut consommer a, alors il évolue tandis que $\mathcal{S}$ reste inchangé ;
- règle *cp4* : si $\mathcal{S}$ peut évoluer en laissant passer le temps, alors l'ensemble de la composition évolue en laissant passer le temps.

$$\frac{(s, \mathcal{S}, B_2) \overset{\rightarrow}{\sigma} (s', \mathcal{S}, B_2')}{< (C, B_1)|(s, \mathcal{S}, B_2) > \overset{null_e}{\underset{\sigma}{\longrightarrow}} < (C, B_1.\sigma)|(s', \mathcal{S}, B_2') >} [\text{cp1}]$$

$$\frac{(C, B_1) \overset{a!}{\longrightarrow} (C', B_1')}{< (C, B_1)|(s, \mathcal{S}, B_2) > \overset{a}{\underset{null_\sigma}{\longrightarrow}} < (C', B_1')|(s, \mathcal{S}, B_2.a) >} [\text{cp2}]$$

$$\frac{(C, B_1) \overset{a?}{\longrightarrow} (C', B_1')}{< (C, B_1)|(s, \mathcal{S}, B_2) > \overset{null_e}{\underset{null_\sigma}{\longrightarrow}} < (C', B_1')|(s, \mathcal{S}, B_2) >} [\text{cp3}]$$

$$\frac{(s, \mathcal{S}, B_2) \overset{t}{\underset{\sigma}{\rightarrow}} (s', \mathcal{S}, B_2')}{< (C, B_1)|(s, \mathcal{S}, B_2) > \overset{t}{\underset{\sigma}{\rightarrow}} < (C, B_1)|(s', \mathcal{S}, B_2') >} [\text{cp4}]$$

Figure 4. Règle sémantique de la composition d'un contexte CDL et d'un système

Notons que la composition de fermeture entre un système et son contexte est assimilable à une composition parallèle asynchrone : les comportements de C et de $\mathcal{S}$ sont entrelacés et la communication est réalisée par échanges asynchrones via des *buffers*.

On note $< (C,B)|(s,\mathcal{S},B') > \not\rightarrow$ pour exprimer que le système et son contexte, associés à leurs *buffers* respectifs, ne peuvent plus évoluer selon les règles ci-dessus (le système fermé est bloqué ou le contexte est terminé). Nous définissons alors l'ensemble des traces (appelées des *runs*) du système fermé par son contexte et à partir d'un état s, par la fonction suivante :

$$[\![C \mid (s,\mathcal{S})]\!] \stackrel{\text{def}}{=} \{a_1 \cdot \sigma_1 \cdot \ \ldots \ a_n \cdot \sigma_n \cdot end_C \mid$$
$$< (C, null_\sigma) \mid (s, null_\sigma) > \quad \xrightarrow[\sigma_1]{a_1} \quad < (C_1,B_1) \mid (s_1,\mathcal{S},B_1') >$$
$$\xrightarrow[\sigma_2]{a_2} \ \ldots \ \xrightarrow[\sigma_n]{a_n} \quad < (C_n,B_n) \mid (s_n,\mathcal{S},B_n') > \quad \not\rightarrow \}$$

$[\![C|(s,\mathcal{S})]\!]$ est l'ensemble des exécutions du système $\mathcal{S}$ fermé par le contexte C et en considérant s comme l'état de départ de $\mathcal{S}$.

Notons que, les contextes étant formés de compositions séquentielles ou parallèles de MSC finis, les traces des contextes sont donc nécessairement finies. En effet, les contextes, lors de la compilation, sont dépliés en contextes simples (mais plus longs) sans aucune boucle. En conséquence, les *runs* du système fermé par son contexte sont nécessairement finis. Nous étendons chaque *run* de $[\![C \mid (s,\mathcal{S})]\!]$ par un événement terminal spécifique end_C permettant à l'observateur de détecter la fin d'un scénario et de "tester" d'éventuelles propriétés de vivacité (une propriété du type "un jour a" se traduira sous cette restriction de finitude des contextes par "un jour a avant end_C").

3.4 Formalisation des observateurs

La formalisation de CDL concerne également l'expression des propriétés permise par le langage. Nous considérons, dans cet article, que les propriétés sont modélisées comme des observateurs. Un observateur est un automate qui observe l'ensemble des évènements échangés entre le système $\mathcal{S}$ et son contexte C (et les évènements survenant dans une trace (*run*) de $[\![C|(init,\mathcal{S})]\!]$) et qui produit un évènement *reject* quand la propriété devient fausse.

Nous considérons, dans la suite, qu'un observateur est un automate émettant un seul évènement de sortie (*reject*), pouvant capter des évènements, produits et reçus par le système et son contexte. Toutes les transitions étiquetées *reject* arrivent dans un état spécifique d'erreur (état de rejet).

Sémantique. Soit un système $\mathcal{S}$ et un contexte C. Soit un observateur $\mathcal{O}$. $\mathcal{S}$ dans l'état $s \in \Sigma$ fermé par C satisfait $\mathcal{O}$, noté $C|(s,\mathcal{S}) \models \mathcal{O}$, si et seulement si aucune exécution de $\mathcal{O}$ confrontée aux runs r de $[\![C|(s,\mathcal{S})]\!]$ ne produit l'événement *reject*. C'est-à-dire :

$$C \mid (s,\mathcal{S}) \models \mathcal{O} \iff \forall r \in [\![C \mid (s,\mathcal{S})]\!],$$
$$(init_o, \mathcal{O}, r) \xrightarrow[null_\sigma]{} (s_1, \mathcal{O}, r_1) \xrightarrow[null_\sigma]{} \ldots \xrightarrow[null_\sigma]{} (s_n, \mathcal{O}, r_n) \not\rightarrow$$

Remarque. Exécuter $\mathcal{O}$ sur un run r de $[\![C|(s,\mathcal{S})]\!]$ revient simplement à remplir le *buffer* de $\mathcal{O}$ par r, et dérouler son exécution à partir de son état initial. Si la propriété est satisfaite, alors cette exécution n'émettra que des évènements vides ($null_\sigma$) (et donc jamais *reject*).

Références