Comprendre et analyser le SAE

4.1 Connaissances liées au model checking

Dans cette section nous définissons les connaissances liées au model checking, que nous abrégerons en *connaissances techniques*, comme l'ensemble des informations requises et produites par le processus de vérification par model checking, telles que les propriétés, le modèle formel, le LTS ou encore les traces. Ces connaissances sont toujours présentes lors de l'activité de diagnostic.

4.1.1 Concepts

SAE

Le système à l'étude (SAE) est décrit par une spécification comprenant un lot de propriétés formelles, et un modèle formel de son comportement.

Composant et processus

Le comportement du système est décrit par un *composant*. Il contient des variables globales (fifos par exemple) et des instances de *processus* reliés entre eux par des *canaux*. Un *processus* est une machine à état qui évolue de manière concurrente aux autres processus. Il est décrit par un ensemble de variables et d'un automate, lui-même composé d'*états* et de *transitions*. Durant l'exécution d'une transition, différentes actions sont réalisées comme l'affectation de variable, la lecture ou l'écriture dans un canal. Un des états est initial.

Message et canal

Un processus communique avec d'autres processus, soit de façon synchrone, soit de façon asynchrone. Dans la thèse, nous faisons le choix que les échanges sont uniquement asynchrones. Dans ce cas précis, des *messages* sont échangés via des *canaux* de type *FIFO* où l'ordre d'envoi des messages est préservé lors de la réception.

Environnement

L'environnement communique également par envoi de messages asynchrones avec le système. Il guide ainsi l'exécution du système suivant un scénario, réduisant de ce fait son espace d'état. Au moment de l'exécution, il est instancié automatiquement sous la forme d'un processus unique.

Graphe d'exploration

Un graphe d'exploration produit par un model checker (OBP dans notre cas) représente l'ensemble des états atteignables du modèle nommés *configurations*, composé du système (modélisé en langage FIACRE) et de l'environnement (décrit avec CDL pour OBP). Un graphe d'exploration est produit par le model checker, celui-ci possède toujours une configuration initiale et un ensemble de configurations finales.

Configuration

Une configuration est une structure de données regroupant des informations relatives à un état du système et son environnement. Une configuration regroupe les informations sur des instances de processus et de leurs informations (variables, états), des canaux de communications, des prédicats ou des observateurs.

Transition

Une transition est un arc entre deux configurations. Chaque transition est associée à un label. Dans le cas d'OBP il représente soit une émission de message, soit une réception de messages, soit un rendez-vous, soit un "internal" (changement interne à un processus).

Trace

Une trace est la description d'un chemin dans le graphe d'exploration. Une trace possède une configuration initiale, et une configuration finale, correspondant généralement au rejet d'un observateur. Il peut exister plusieurs traces ayant la même configuration initiale et finale.

Prédicats

Un prédicat permet d'évaluer des expressions basées sur les données d'exécution du modèle (variables, états...). Par exemple, déterminer si le processus *NET* est dans l'état *idle*, se traduit en CDL par *predicate* P_1 *is* $Net1@Idle$. Déterminer si la variable *length* de la fifo F_1 du composant Run_1 est inférieure à une constante donnée *SIZE*, se traduit en CDL

par *predicate* P_2 *is* $Run1 : F_1.length \ <= \ SIZE$. Autre cas de figure, un événement associé à un changement de valeur d'un prédicat. Exemple, l'événement E_1 survient lorsque le prédicat P_1 devient vrai, se traduit en CDL par l'expression E_1 *is* P_1 *becomes true*.

Observation des propriétés

Un observateur, au sens OBP, est un automate. Selon la séquence des transitions, consécutive à l'observation des activités du système et de l'environnement, l'observateur atteint soit un état que l'on souhaite obtenir (état succès), soit un état qui ne doit pas apparaître (état rejet). L'observateur peut observer des occurrences d'événements (émission ou réception de message), ou des changements de valeur booléenne d'un prédicat. Les transitions, déclenchées par l'observation des événements (*event*), peuvent être gardées par des prédicats (*predicate*). Une version simplifiée d'une transition (non-temporisée) entre un état source (*sourceState*) et un état cible (*targetState*) est définie grâce à la syntaxe *sourceState – / predicate / event / -> targetState*. Un observateur permet d'exprimer une propriété de type sûreté (il contient un état reject).

Logique temporelle

La logique temporelle, comme la LTL, permet d'exprimer des propriétés et des prédicats logiques à l'aide d'opérateurs logiques (nous utiliserons ici *not* ($\neg$), *and* ($\wedge$), *or* ($\vee$) et *implies* ($\rightarrow$)) et temporels (nous utiliserons ici *globally* ($\square$), *eventually* ($\lozenge$) et *weakly until* (W)). Pour exprimer de telles propriétés, il est possible d'utiliser Plug, une extension à OBP. Si les observateurs permettent d'exprimer pleinement les propriétés de sûreté, ils ne permettent pas d'exprimer la vivacité. Il faut donc utiliser la LTL pour exprimer des propriétés de vivacité.

4.1.2 Modèle

Reprenons le premier scénario. Le modèle à vérifier est exprimé dans le langage formel Fiacre. L'architecture est composée de 4 entités, un contrôleur global appelé *GC*, deux contrôleurs locaux Plc_1 et Plc_2 et un réseau de communication *Network* reliant le *GC* aux Plc_1 et Plc_2. Les contrôleurs locaux permettent d'accéder à une ressource partagée *Res*, chacun par un accès, respectivement Res_1 et Res_2.

L'architecture fonctionne de la manière décrite par les automates 4.1, 4.2 et 4.3. Une entité de l'environnement souhaite accéder à une ressource de l'architecture. Elle envoie un message au *GC* indiquant dans le message la demande d'accès à la ressource (Res_1 ou Res_2) et l'opération à réaliser sur cette ressource (*Read* ou *Write*). *GC* reçoit cette requête (état *Idle*), l'ignore si elle lui est directement adressée (état *ReceivedForward*, puis transition *GcIgnoring*), ou dans le cas contraire, la retransmet à un *PLC* via le

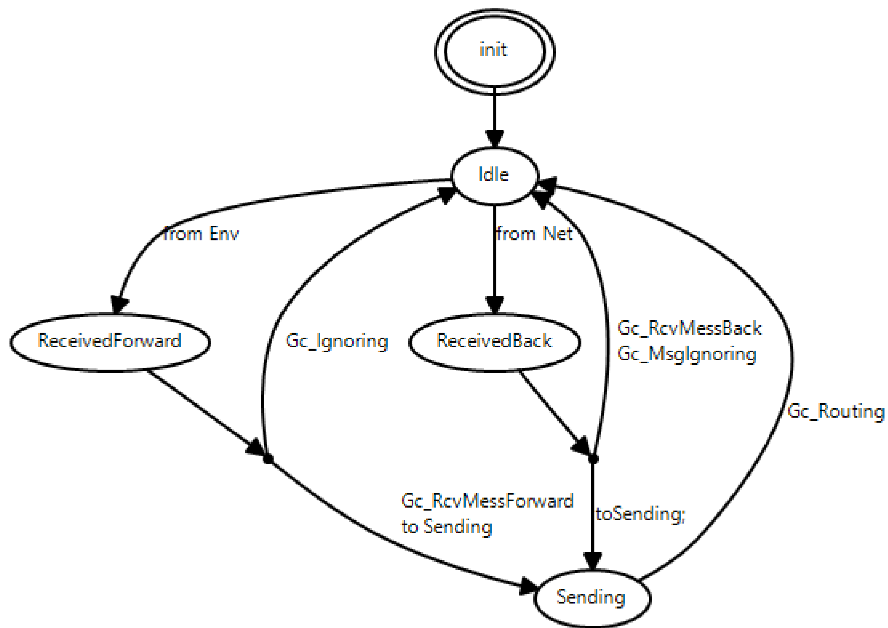

FIGURE 4.1 – Automate de GC

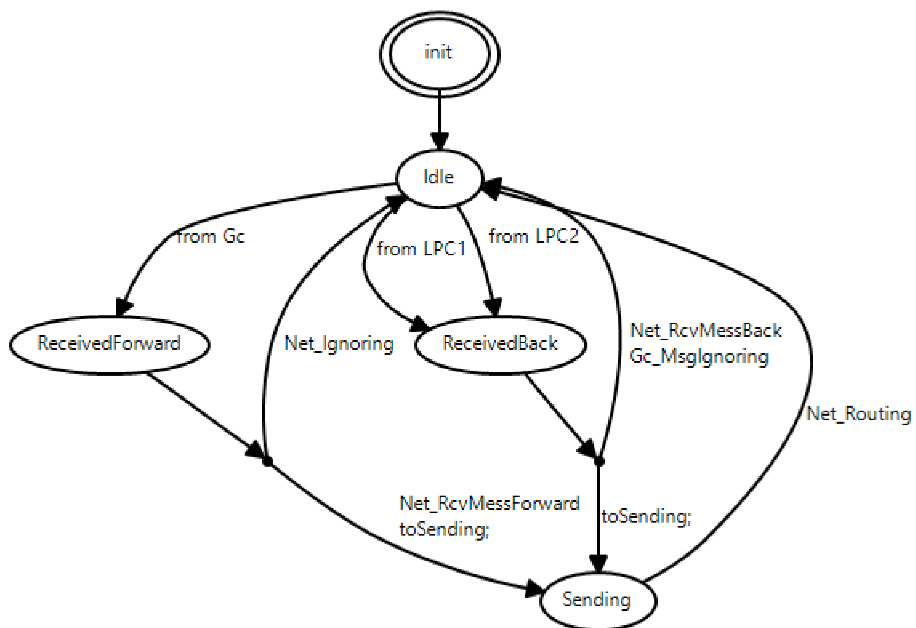

FIGURE 4.2 – Automate de NET

Network (état *Sending*). Le *Network* traite ensuite la requête (état *ReceivedForward*). Comme *GC*, si la requête lui est destinée, il l'ignore (retour dans l'état *Idle*), sinon il la retransmet au *PLC* concerné. Le *PLC* reçoit la requête (état *Received*), accède à la ressource (état *Access*) et envoie un message contenant une réponse (état *Sending*). La réponse est transmise à l'environnement en passant d'abord par le *Network*, puis par le *GC*, via leurs états respectifs (*ReceivedBack*).

Les entités considérées jouent des rôles différents. Le rôle PLC_1_OWNER a les droits en lecture et écriture sur RES via Res_1 mais pas via Res_2. Le rôle PLC_2_OWNER a les droits en lecture et écriture via Res_2 mais pas via Res_1. Nous considérons deux clients, le $Client_1$ joue le rôle de PLC_1_OWNER, et le $Client_2$ joue le rôle de PLC_2_OWNER.

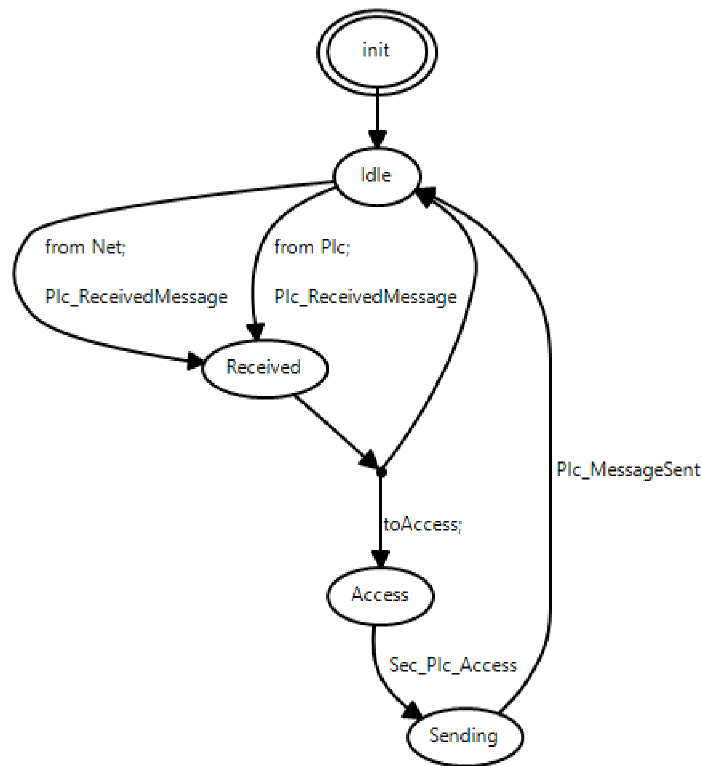

FIGURE 4.3 – Automate de PLC

Les processus sont reliés par des FIFOs de taille finie. Ici la taille initiale est 4, c'est à dire que la fifo peut contenir jusqu'à 4 messages, pas au delà. Les FIFOs supportent des messages de deux types, *SEND* (requête) ou *RECV* (réponse). Un message contient différents paramètres, l'entité source à l'origine du message (*source*), l'entité destinataire (*target*), et une donnée (*data*). Une donnée (*data*) est composée de trois informations, le type d'opération, soit lecture (*RED*) ou écriture (*WRT*), l'accès à la ressource, soit Res_1 ou Res_2, et un type d'acquittement, soit *ACK* ou *NACK*. Le listing 4.1 présente un exemple de message exprimé en Fiacre.

```
1 // Exemple, un message de lecture par le Client1 de la Res1 geree par le
    PLC1
2 CLT1_READ_PLC1 is {source = ID_CLT_1, target = ID_PLC_1, data = RED_RES1}
```

Listing 4.1 – Message

Pour simplifier, les messages sont associés à une constante qui respecte des conventions de nommage dans l'ordre suivant *type de message [envoi (SEND) | réception (RECV)], entité source, entité cible, type d'accès [lecture (RED) | écriture (WRT)], ressource.* Par exemple, une demande de lecture de la Res_1 détenue par le Plc_1 par le $Client_1$ est exprimée par la constante $SEND_CLT_1_PLC_1_RED_RES_1$.

4.1.3 Propriétés

Prenons par exemple deux propriétés à vérifier pour ce modèle, une de vivacité et une de sûreté. La première propriété, $pty_CLT_1_PLC_1_RED_RES_1_Scenario_1a$, exprime que l'envoi du message $CLT_1_PLC_1_RED_RES_1$ par l'entité Clt_1 est suivi d'une réception de Clt_1 du message $CLT_1_PLC_1_RED_RES_1_ACK$. Il est possible d'exprimer une telle propriété en LTL :

$$\boxed{\begin{array}{l} pty_CLT_1_PLC_1_RED_RES_1_Scenario_1a: \\ \Box\{SEND_CLT_1_PLC_1_RED_RES_1 \rightarrow \Diamond RECV_PLC_1_CLT_1_RED_RES_1_ACK\} \end{array}} \tag{4.1}$$

Avec la syntaxe CDL, dans l'outil OBP Plug sous la forme suivante :

```
1 pty_CLT1_PLC1_RED_RES1_Scenario1a is {
2       [] (| SEND_CLT1_PLC1_RED_RES1 |=> <>| RECV_PLC1_CLT1_RED_RES1_ACK | )
3 }
```

Une seconde propriété affine la première propriété, en exigeant que si le client demande une seule fois la ressource il ne peut recevoir qu'un seul acquittement. Cette propriété, de sûreté cette fois, est exprimée en CDL par l'observateur suivant. Celui-ci rentre dans un état de rejet si deux demandes de ressources sont réalisées, ou si deux acquittements sont reçus.

```
1 property pty_CLT1_PLC1_RED_RES1_Scenario1b is
2 {
3   start -- / / send_CLT_1_PLC_1_RED_RES1      / -> wait;
4   wait  -- / / recv_CLT_1_PLC_1_RED_RES1_ACK  / -> onlyonce;
5   wait  -- / / send_CLT_1_PLC_1_RED_RES1   / -> reject;
6   onlyonce -- / / recv_CLT_1_PLC_1_RED_RES1_ACK / -> reject
7 }
```

Listing 4.2 – Propriété pty_CLT1_PLC1_RED_RES1_Scenario1b

4.1.4 Exploration

Reprenons le premier scénario évoqué dans la section 2.1.2. Luka reçoit ce modèle réalisé par un tiers. Il est en charge de sa vérification, et s'il n'est pas expert dans le domaine des SCADA, il possède une expérience significative en vérification par model checking. Ceci sous entend qu'il s'appuie sur les connaissances liées au model checking définies dans ce chapitre.

Luka suit l'approche décrite par [BK08]. Ainsi, à l'aide du model checker OBP, il génère le LTS correspondant au modèle, et l'explore. Afin de limiter l'explosion combinatoire, Luka effectue des vérifications cadrées par des scénarios différents, formalisés sous la forme de contextes CDL. Après vérification, il obtient un graphe d'exploration qu'il analyse. Il y découvre que la propriété $pty_CLT_1_PLC_1_RED_RES_1_Scenario_1a$ est violée, indiquant que la demande d'accès à une ressource par le client ne lui renvoie jamais d'acquittement (ACK).

Luka démarre une phase de diagnostic en étudiant un contre-exemple. Dans OBP, une trace est divisée en deux parties, la liste des configurations et la liste des transitions qui composent le chemin passant par ces configurations, un extrait est présenté sur le listing 4.3.

```
Config: 0{
 component: '{Run}1' [FIFOs]
 proc: '{Gc}1' [@Idle, targetAvail=false, canPass=false, mess={} ]
 proc: '{Net}1' [@Idle, targetAvail=false, canPass=false, mess={} ]
 proc: '{Plc}1' [@Idle, resAvail=false, access=false, opRes={}, mess={} ]
 proc: '{Plc}2' [@Idle, resAvail=false, access=false, opRes={}, mess={} ]
 context: '{env}1' [@start]
 observer: '{pty_CLT1_PLC1_RED_RES1_Scenario1b}1' [@start]
}

transitions: {
( 0 , "internal" , 1)
( 1 , "internal" , 2)
   ...
( 5 , "internal" , 6)
}
```

Listing 4.3 – Trace du scénario 1

Luka est confronté à deux enjeux principaux, la taille de la trace et son niveau de détail. En conséquence, pour établir un diagnostic Luka va devoir réaliser différentes activités cognitives successives.

4.2 Activités cognitives de diagnostic des traces

Cette section présente des activités cognitives mobilisées dans le diagnostic.

4.2.1 Première tentative

Isolation et visualisation

Dans un premier temps il cherche à analyser la progression des messages en isolant les communications entre les processus. Pour comprendre plus aisément les communications, il opte pour l'utilisation d'un outil de visualisation de trace fourni par le model checker OBP. La représentation choisie est un diagramme de séquence présenté sur la figure 4.4, mettant en avant les interactions entre les processus. Dans le diagramme chaque ligne de vie correspond à un processus. Entre deux lignes de vies, représentés par des traits verticaux, figurent les messages échangés. Les traits pointillés horizontaux représentent des changements d'états internes aux processus.

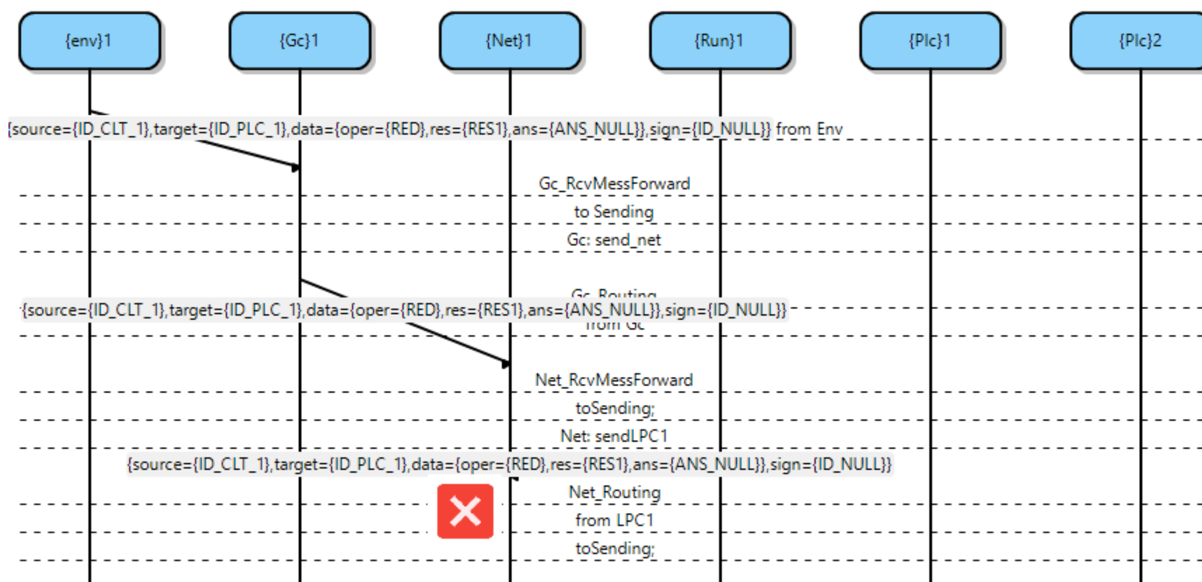

FIGURE 4.4 – Diagramme de séquence de la trace

Cette visualisation permet d'identifier aisément qu'un message n'est pas transmis entre deux processus, *Net* et Plc_1. Deux hypothèses sont envisageables, le message n'est jamais transmis, ou bien il n'est pas transmis au bon destinataire.

Localisation

Il faut maintenant analyser les instructions suspectes. Luka cherche alors à localiser dans le programme Fiacre l'ensemble des instructions jouant un rôle dans la communication (envois et réceptions de messages, déclarations des canaux). Il remarque qu'il n'y a pas de connexion entre Plc_1 et Net, l'instanciation du processus Net étant mal réalisée comme le montre le listing 4.4.

```
par
Gc (ID_Gc, ...) ||  Plc (ID_PLC_1, ...) ||  Plc (ID_PLC_2, ...) ||
//  Il n'y a pas de FIFO entre NET et PLC1, uniquement PLC1 vers NET
|Net  (ID_NET, &fifo_Gc_NET, &fifo_PLC_1_NET, &fifo_PLC_2_NET, &fifo_NET_Gc
    , &fifo_PLC_1_NET, &fifo_NET_PLC_2)
end
```

Listing 4.4 – Premier problème

Ici les activités cognitives de Luka ont consisté à isoler les communications entre processus à travers des visualisations de traces, puis localiser les instructions suspectes.

4.2.2 Seconde tentative

Visualisation

Une fois la correction apportée, Luka évalue sa qualité en effectuant une autre vérification. Le graph d'état atteint désormais 1 871 états et 5 470 transitions, et devient difficilement compréhensible alors qu'OBP signale une nouvelle fois qu'une propriété est violée, comme le montre le résultat extrait d'OBP 4.5.

Il s'agit de $pty_CLT_1_PLC_1_RED_RES_1_Scenario_1b$ laissant supposer que deux messages d'acquittements sont transmis au *Client* alors qu'une seule une demande est effectuée.

```
pty_CLT1_PLC1_RED_RES1_Scenario1a: reached success in conf 25.
pty_CLT1_PLC1_RED_RES1_Scenario1b: reached reject in conf 95.
```

Listing 4.5 – Vérification du scénario 1b

De la même manière que précédemment, Luka observe une trace à travers un diagramme de séquence contenant la configuration 95. Le message est transmis entre le $Client_1$ et Plc_1, puis un acquittement est renvoyé par Plc_1 au $Client_1$. Mais l'acquittement est renvoyé une seconde fois, comme le montre le diagramme 4.5.

Stimulation

Ici, le diagramme de séquence ne permet pas d'aller plus loin dans l'analyse, il faut changer de stratégie. Expert en model checking, Luka tente d'analyser le modèle en le

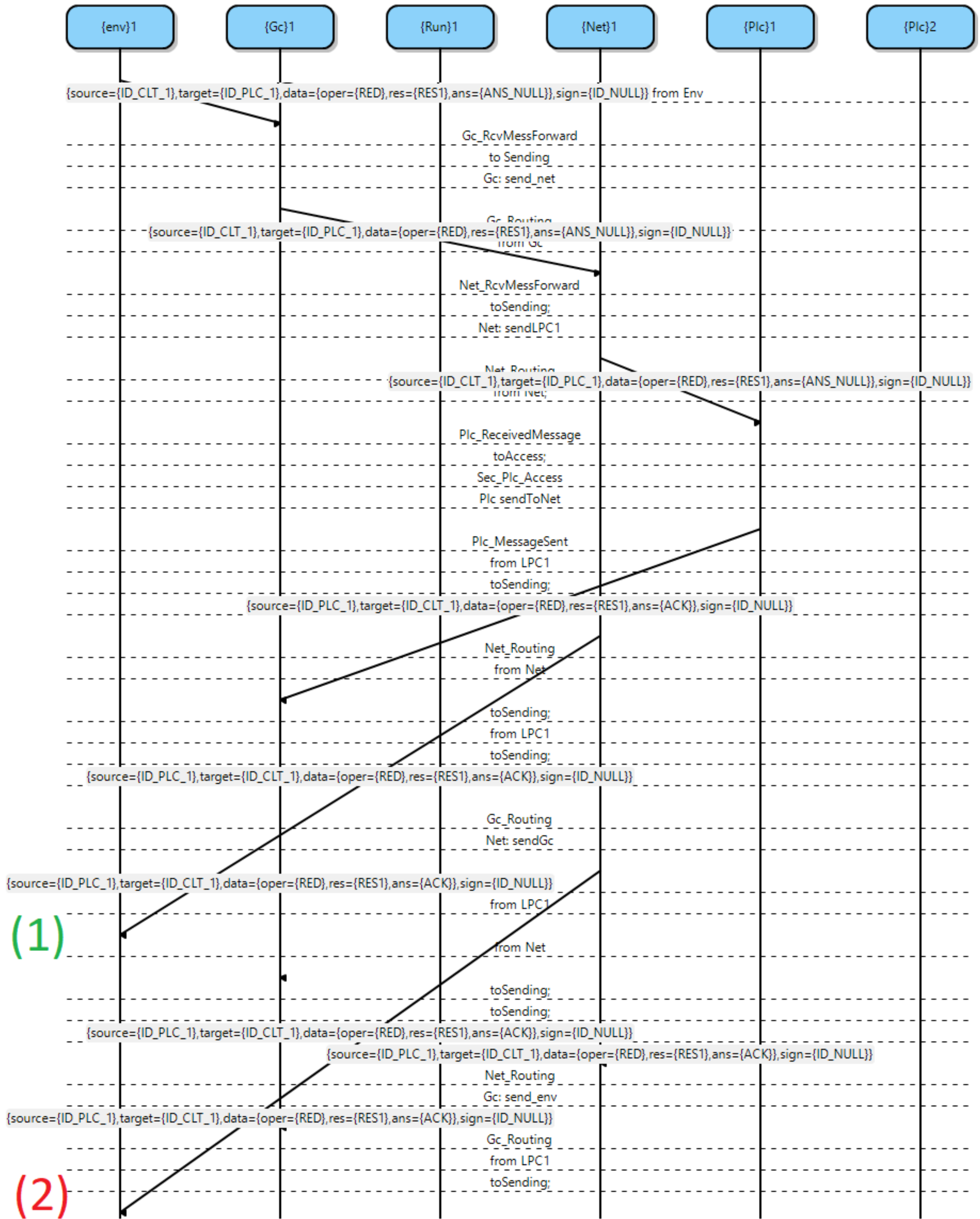

FIGURE 4.5 – Diagramme de séquence de la trace, deux acquittements sont renvoyés

stimulant. Il modifie par exemple la taille des FIFOs, qui passent de 4 à 2. Une nouvelle vérification indique une réduction de l'espace d'état, 677 états et 1 794 transitions, et que la propriété est toujours violée. Les FIFOs ont une certaine tendance à se remplir, elles représentent un point critique à observer. Stimuler le modèle permet d'analyser le fonctionnement du système et ses limites.

Génération

Luka applique de nouvelles propriétés, que nous appelons *propriétés de contrôle*, lui permettant d'observer des comportements particuliers des FIFOs suspectées. Il les énonce en CDL (listing ci-dessous). Le prédicat *pre_Fifo_push* est vrai si la FIFO PLC_1_NET contient un message, et le prédicat pre_Fifo_pop est vrai si la FIFO n'en contient pas. Deux événements dépendent de ces prédicats, evt_Fifo_push est déclenché quand le prédicat pre_Fifo_push est vrai, et evt_Fifo_pop est déclenché quand le prédicat pre_Fifo_pop est vrai.

Un nouvel observateur, $pty_CLT_1_PLC_1_RED_RES_1_Scenario_1bFifo$, est en succès s'il observe successivement les événements evt_Fifo_push puis evt_Fifo_pop. Autrement dit si cet observateur n'est jamais en succès, la FIFO n'est jamais vidée.

```
1  predicate pre_Fifo_push is {{Run}1:fifo_PLC_1_NET.length = 1} }
2  predicate pre_Fifo_pop is {{Run}1:fifo_PLC_1_NET.length = 0} }
3
4  event evt_Fifo_push is {pre_Fifo_push becomes true}
5  event evt_Fifo_pop is {pre_Fifo_pop becomes true}
6
7  property pty_CLT1_PLC1_RED_RES1_Scenario1bFifo is{
8    start  -- / / evt_Fifo_push /  -> wait;
9    wait  -- / / evt_Fifo_pop / -> success
10 }
```

Listing 4.6 – Propriété pty_CLT1_PLC1_RED_RES1_Scenario1bFifo et prédicats

Luka affecte la taille maximale des FIFOs à 1, par conséquent, il ne peut pas y avoir plus d'une fois l'événement evt_Fifo_push, ou autrement dit, une FIFO remplie n'a pas d'autre choix que d'être vidée. Le résultat de la nouvelle exploration donne :

```
1 pty_CLT1_PLC1_RED_RES1_Scenario1a:  reached success the first time in conf
    25.
2 pty_CLT1_PLC1_RED_RES1_Scenario1b:  reached reject the first time in conf
    89.
3 Violated asserts: 'pre_Fifo_push the first time in configuration 21.
```

Listing 4.7 – Vérification du scénario 1b

L'observateur de la FIFO atteint l'état *wait* (après observation de evt_Fifo_push) mais ne parvient pas à l'état *success* (n'observe pas d'événement evt_Fifo_pop). Premier

diagnostic, la FIFO a bien été remplie par un message, mais n'est pas dépilé par la suite. Le prédicat pre_Fifo_push à *false* confirme qu'elle contient bien un message à partir de la configuration 21. Luka fait l'hypothèse que la FIFO n'est pas dépilée. Il modifie le modèle pour dépiler la FIFO, et une nouvelle tentative de vérification vérifiant la propriété confirme l'hypothèse, le diagnostic est que la FIFO n'était jamais dépilée.

4.3 Conception d'une nouvelle solution

Après avoir compris que la FIFO n'était pas dépilée, Luka a corrigé le modèle et la propriété est désormais satisfaite. Une nouvelle propriété est souhaitée, $P_NotRes_1AndRes_2$, qui indique que la ressource *Res* ne peut pas être accédée en même temps par les deux *Plcs*, autrement dit il n'est pas possible que Plc_1 et Plc_2 soient simultanément dans leur état *Access*. Cette propriété est exprimée en LTL par :

$$\boxed{P_NotRes_1AndRes_2 : \Box\neg\{PLC_1@Access \wedge PLC_2@Access\}} \tag{4.2}$$

Après une première vérification, la propriété $P_NotRes_1Res_2$ est violée. Luka ne fait pas d'autres analyses, les connaissances liées au model checking ne lui permettent pas d'aller plus loin, il doit apporter des modifications à la solution, et changer de point de vue pour passer du correcteur au concepteur.

4.3.1 Solution par mécanisme de drapeaux

Abstraction

Il imagine alors un mécanisme de drapeaux que chaque processus devra lever pour signaler son désir d'accéder à *Res*. L'algorithme imaginé est le suivant :

```
1  Plc1:
2    while true {
3      while flagPlc2 == up {};
4      flagPlc1 := up;
5      access;
6      flagPlc1 := down;
7    }
8
9  Plc2:
10   while true {
11     while flagPlc1 == up {};
12     flagPlc2 := up;
13     access;
14     flagPlc2 := down;
15   }
```

Les drapeaux sont deux variables booléennes partagées entre les deux *Plcs* concurrents, nommées $flagPlc_1$ et $flagPlc_2$. Quand ils sont levés (leur valeur est UP), ils indiquent que le *Plc* souhaite accéder à la ressource, à l'inverse quand ils sont baissés (leur valeur est $DOWN$), ils indiquent que le *Plc* a libéré la ressource. Par exemple, $flagPlc_1 := UP$ signifie que le Plc_1 lève son drapeau, et donc qu'il souhaite accéder à la ressource. Un test est effectué avant de lever le drapeau afin de vérifier que le drapeau adverse n'est pas levé, auquel cas il faut attendre qu'il se baisse.

Implémentation

Pour intégrer cet algorithme au système, Luka doit implicitement le lier au modèle (Fiacre en l'occurrence). Par exemple l'opération $flagPlc_1 == UP$ est mise en œuvre dans la transition entre les états de l'automate *Received* (symbolisé par deux sous états, *Received_Test* et *Received_Affect*), et *Access*. Après intégration il obtient le comportement décrit par la machine à états de la figure 4.6.

Les transitions entre états respectent la notation $Si \xrightarrow{\{Event\}[Condition]} Sj$ où *Si* et *Sj* sont des *états*, la flèche représente une *transition* déclenchée sur l'évaluation d'une *condition* booléenne et/ou la réception d'événements (*events*, précédés de ?). L'*action* d'un état est une instruction (affectation de variable ou un événement émis (précédé de !)). Dans la sémantique du langage Fiacre, l'ensemble des actions exécutables d'un état est considéré atomique.

A l'initialisation du système, les deux drapeaux sont baissés, et les *Plcs* sont dans leur état $Idle$. Ils attendent un message provenant du processus Net via la fifo $fromNetPlc$. Si un *Plc* reçoit un message du Net, il progresse dans l'état *Received* (comprenant l'état *Received_Test* et l'état $Received_Affect$) pour le traiter. Si le message correspond à une demande d'accès à la ressource, le *Plc* doit s'assurer que son *Plc* concurrent ne demande pas lui aussi la ressource (par exemple Plc_2 demande la ressource quand $flagPlc_2$ est à UP), auquel cas il devra attendre dans l'état $Received_Test$. Si le processus concurrent ne demande pas la ressource, alors le processus lève son drapeau (par exemple $flagPlc_1 := UP$ pour Plc_1) et accède à la ressource (état *Access*). Lorsqu'il a terminé il sort de la section critique (état *Sending*) en baissant son drapeau (par exemple $flagPlc_1 := DOWN$ pour Plc_1) pour informer de la disponibilité de la ressource. Enfin il transmet une information acquittant le processus Net via la fifo $fifoPlcNet$.

4.3.2 Vérification et diagnostic

Après vérification, il s'avère que la propriété $P_NotRes_1AndRes_2$ est violée, signalant l'existence d'un contre exemple où Plc_1 et Plc_2 utilisent en même temps la ressource, ce qui déclenche une phase de diagnostic.

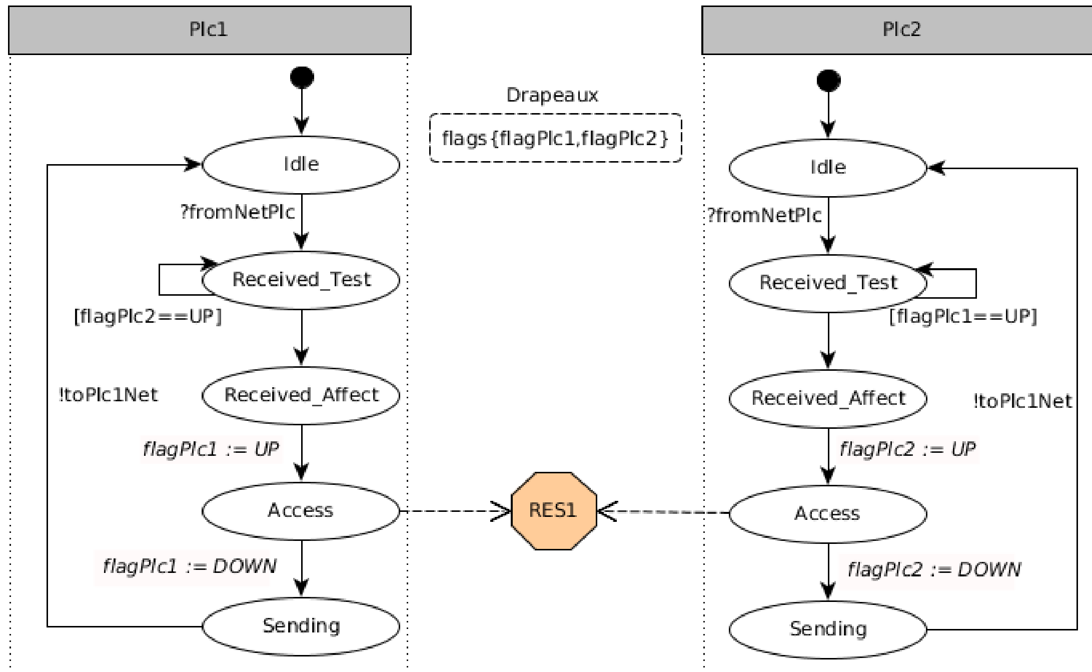

FIGURE 4.6 – Comportement des Plcs avec un mécanisme de drapeaux.

Réduction

Un contre-exemple brut (a), filtré (b) et vu de l'algorithme (c) est présenté dans la figure 5.1. Après une étude infructueuse du contre-exemple brut (a) de façon analogue au scénario précèdent (visualisations, expression de propriétés de contrôle), il s'avère que l'algorithme est bien implémenté mais mal conçu. Le contre-exemple brut (a) contient de nombreuses configurations dont quelques unes uniquement sont liées à l'algorithme mis en cause, or si le problème vient de l'algorithme, il faut permettre de raisonner dessus.

Luka réduit alors le contre-exemple aux seules configurations produites par l'action de l'algorithme (b). Ces configurations regroupent des informations techniques (canaux, processus, états...), et il est difficile de les percevoir sous l'angle de l'algorithme (*while, flag :=up*). C'est le problème du fossé sémantique. Pour parvenir à le réduire, Luka doit extraire du contre-exemple les informations liées à cet algorithme en explicitant les liens implicites qui ont permis de passer de l'algorithme au modèle.

Interprétation

Pour résoudre le fossé sémantique entre l'algorithme et le contre-exemple, Luka doit disposer d'une nouvelle vue (c) qui lui permettrait de raisonner sur l'algorithme. Cette vue n'existe pas dans OBP, mais la possibilité d'annoter les transitions peut résoudre en partie

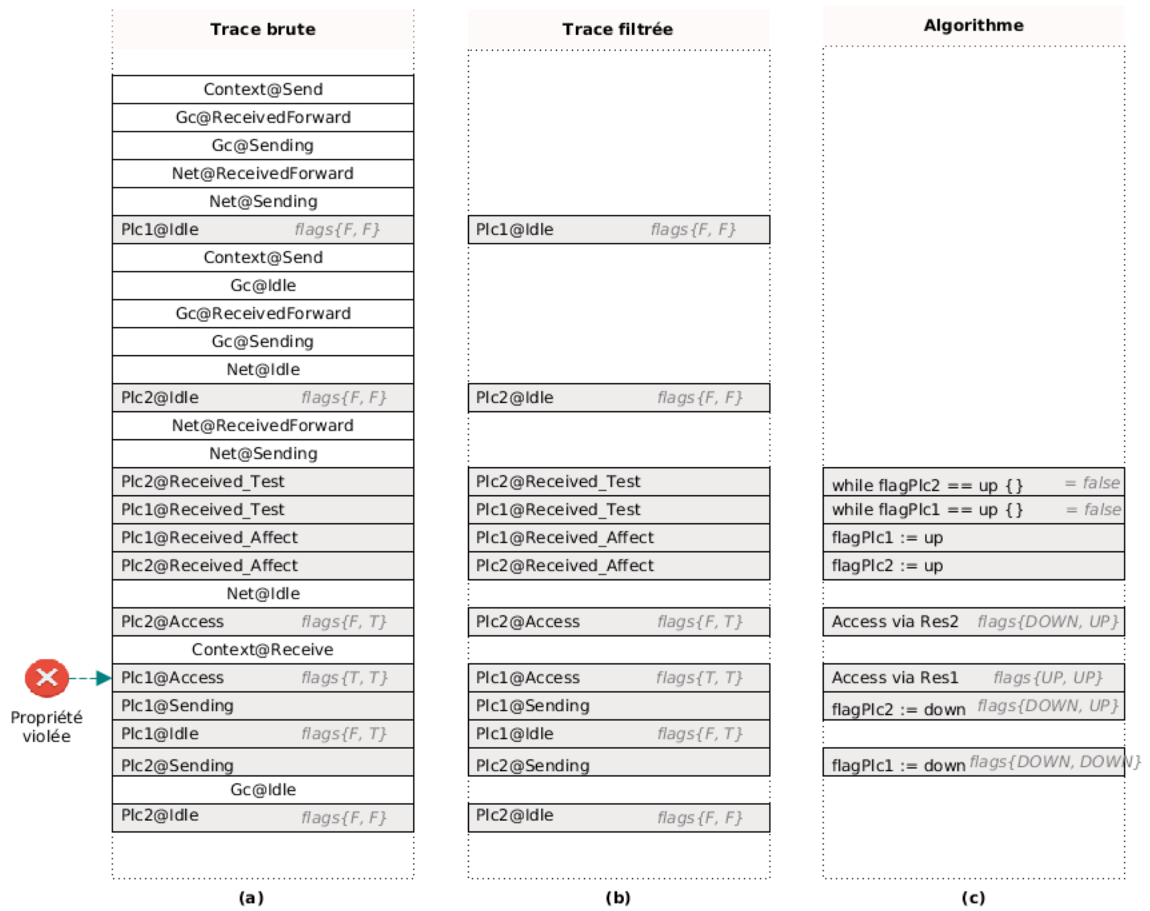

FIGURE 4.7 – Un contre-exemple brut (a), filtré (b) et vue de l'algorithme (c)

le problème. En complément, nous proposons d'utiliser un mécanisme de fédération de modèles, décrit à la section 7.2.2, pour établir des liens sémantiques entre traces, modèles et algorithmes.

Évaluation et correction

L'analyse de la trace décrit la situation suivante : les deux drapeaux sont initialement baissés. Le premier processus vérifie que le drapeau adverse est baissé ; donc lève son drapeau et entre dans l'état *Access*. Ceci est entrelacé avec un comportement similaire du deuxième processus. Ce qui explique pourquoi la propriété est violée, Luka doit revoir la solution et s'intéresser au domaine de l'exclusion mutuelle.

4.4 Conclusion

4.4.1 Cadre conceptuel

Luka doit donc mettre en œuvre un ensemble d'activités cognitives, chacune apparentée à un outil ou technique. Ces activités peuvent être placées dans la taxonomie de Bloom comme le montre le tableau 4.1. *Se souvenir* consiste à retrouver l'emplacement dans un environnement technique d'une information identifiée. Par exemple Luka est capable de retrouver le modèle, les propriétés ou les contre-exemples. *Comprendre* est la superposition de représentations. Pour Luka, cela consiste par exemple à visualiser les contre-exemples à travers des diagrammes de séquence, ou bien faire le lien entre les éléments du modèle et le contre-exemple, ou les éléments du modèle et l'algorithme. *Appliquer* consiste notamment à générer un ensemble de propriétés de contrôle dans le but de comprendre l'état des FIFOs. Luka réalise différentes *analyses*. Il isole les communications entre les processus, il localise dans le modèles les éléments suspicieux, il stimule le modèle en jouant sur différentes variables ou il réduit la trace selon un point de vue particulier. Les *évaluations* sont d'une part, la vérification par model checking qui donne le statu des propriétés par rapport au modèle, et d'autre part les diagnostics (finaux ou intermédiaires) énonçant la cause de la violation de propriété, qu'elle soit issue du modèle ou du design. Pour finir Luka réalise une activité de *création* lorsqu'il conçoit le mécanisme de drapeau.

4.4.2 Changement de point de vue

A travers le prisme des connaissances liées au model checking, le diagnostic est difficile à établir pour deux raisons principales, le grand nombre d'objets exposés par les traces analysées (nombreuses configurations, transitions, taille des graphes d'exploration), et

	Connaissances liées au model checking (Concepts, Modèle et Propriétés formelles...)
Se souvenir	OBP, Fiacre, Ltl, CDL
Comprendre	Visualiser (MSC), Corréler (Semantic gap)
Appliquer	Générer des propriétés de contrôles et les vérifier
Analyser	Isoler les communications, localiser le code, stimuler, réduire la trace
Évaluer	Vérifications (model checking), Diagnostics (erreurs de modèle ou design)
Créer	Mécanisme de drapeau

Table 4.1 – Les activités cognitives de Luka classées selon Bloom

leur niveau de détail, parfois décorrélés des concepts "métier" (artefacts obtenus dans le cadre des transformations de la chaîne outillée sans rapport direct et évident avec la couche conceptuelle de haut niveau). Il est difficile pour l'utilisateur de pouvoir manipuler l'intégralité de ces données pour analyser son problème.

Rester au niveau des traces ne permet pas toujours de résoudre le problème, et il faut souvent raisonner sur l'algorithme. Ce changement de point de vue sur le contre-exemple contraint le diagnosticien à résoudre un fossé sémantique entre les spécifications et le contre-exemple exprimé par des connaissances liées au model checking. Il doit ainsi expliciter les corrélations entre les spécifications et le modèle qui les implémentent. Ces corrélations peuvent être réalisées techniquement par la mise en œuvre de prédicats, de propriétés, ou parfois d'annotations relatives aux spécifications mais décrites sur la base de connaissances liées au model checking.

L'ingénierie de domaine facilite la transition de la fabrication artisanale d'une solution unique en son genre vers la production automatisée d'une famille de solutions [CE00]. Placer l'application à vérifier au sein d'un domaine éprouvé permet d'effectuer un nouveau changement de point de vue. L'ingénierie de domaine nous aide à concevoir des composants réutilisables et une architecture commune pour une famille de solutions. Pour bénéficier de l'expérience d'un domaine, il faut avoir accès à des problèmes résolus et des problèmes en échec. Il faut pouvoir modifier une solution passée pour l'adapter à une nouvelle situation-problème. Ce sont ces enjeux que nous allons traiter dans le chapitre suivant, où Ciprian mène des activités de conception et de vérification autour du SAE, en bénéficiant de connaissances de domaine.

Chapitre 5

Formaliser, partager et réutiliser les connaissances

5.1 Introduction

On distingue généralement la donnée de l'information, l'information étant une donnée à laquelle on associe du sens. Jusqu'à présent, les activités cognitives de diagnostic manipulaient des informations de nature uniquement technique. Cependant celles-ci restent parfois insuffisantes pour établir un diagnostic, et doivent être complétées par des informations provenant du domaine. Ces informations sont souvent hétérogènes et il n'y a pas de consensus ni sur leur formalisation, ni sur un système permettant leur organisation et leurs interactions. Dans cette section, nous présentons comment les activités de diagnostic bénéficient des informations du domaine, et nous présentons une formalisation permettant leur réutilisation.

5.2 Prise en compte du domaine

Prenons le second scénario où Ciprian joue le rôle de concepteur du système. Ciprian reprend le modèle de Luka où deux processus concurrents Plc_1 et Plc_2 partagent une même ressource Res_1. Cette ressource ne peut pas être accédée au même instant par Plc_1 et Plc_2 (aussi bien en lecture qu'en écriture).

5.2.1 Description du domaine

En explorant la littérature, il y découvre le problème de l'exclusion mutuelle formulée par Lamport dans [Lam86]. Le problème est décrit comme suit : Considérons un processus Pr_i, dont la structure est composée d'une opération non-critique (NCS) et d'une opération critique (CS). Ces opérations sont considérées élémentaires, aussi, ni la structure interne de chaque opération ni même leur durée n'est considérée. Ces opérations sont exécutés alternativement, comme dans l'exemple suivant : $NCS \rightarrow CS \rightarrow NCS \rightarrow CS$....

Supposons deux processus distincts, Pr_i et Pr_j respectant cette structure, l'exclusion mutuelle (entre ces deux processus) revient à dire que les opérations $Pr_iCS \wedge Pr_jCS$ ne

doivent jamais être exécutées simultanément. Cette propriété est nommée ici $PMutexPr_iPr_j$, et est équivalente à l'expression LTL :

$$\boxed{PMutexPriPrj : \Box\neg(Pr_iCS \wedge Pr_jCS)} \tag{5.1}$$

Pour que ce principe soit respecté, il faut ajouter un mécanisme qui empêche cette propriété d'être violée. Celui-ci consiste en un ensemble d'opérations de synchronisation inter-processus. Ces opérations ne doivent pas être concurrentes des sections critiques et non critiques de ce même processus et doivent être atomiques. Ainsi, l'opération $Trying$ réalise une demande d'accès à la ressource, et doit être exécutée entre la section non critique et la section critique. L'opération $Exit$ réalise la libération de la ressource, et doit être exécutée entre la section critique et la section non critique suivante. La structure d'un processus se décrit alors comme :

```
1 initial declaration;
2   repeat forever
3     noncritical section ;
4     trying ;
5     critical section ;
6     exit ;
7   end repeat
```

La séquence d'opérations est la suivante : $NCS \rightarrow Trying \rightarrow CS \rightarrow Exit \rightarrow NCS....$

5.2.2 Application du domaine

Ciprian connaît désormais l'énoncé du problème, l'exclusion mutuelle. Son problème se compose d'une structure faite de concepts abstraits ($NCS, Trying, CS, Exit$), et de propriétés abstraites qui devront être redéfinies au regard de la solution réalisée.

En appliquant le domaine de l'exclusion mutuelle, Ciprian rajoute de nouvelles propriétés à son système, par exemple les deux *Plcs* ne peuvent pas accéder à la ressource critique en même temps. Cette propriété est une *propriété de problème de domaine (PPD)* .

Les *PPDs* pour l'exclusion mutuelle sont abstraites par rapport à une solution choisie. Pour les appliquer, il faut identifier parmi les éléments de la solution, ceux qui correspondent aux éléments des *PPDs*. Ciprian fait la correspondance suivante : - l'opération $Trying$ est réalisée dans la solution à deux instants dans l'état *Received*, lors de l'évaluation du drapeau du processus concurrent (transition entre état *Received_Test* et $Received_Affect$ avec l'opération $[flag == UP]$), et la levée du drapeau (transition entre l'état $Received_Affect$ et $Access$ avec l'opération $[flag := UP]$) ; - l'opération d'accès à la section critique est réalisée entièrement dans l'état $Access$; - enfin l'opération