Vérification de systèmes temps-réel par instrumentation de code

Laure DANTHONY

janvier à mars 2002

Résumé

Dans ce compte-rendu du stage MIM2 réalisé du 15 janvier au 15 mars 2002 au Laboratoire Vérimag, Grenoble, nous présentons les travaux effectués dans le domaine de la vérification des systèmes temps-réel. Après une introduction de la problématique et des notations, nous faisons un bilan des avancées récentes de ce domaine, puis nous abordons le cœur du problème qui consiste à s'éloigner des procédés habituels de Model Checking. Nous rendons compte des travaux effectués autour du langage maison IF, et nous proposons une automatisation de la vérification de certaines propriétés de systèmes décrits en IF, notamment en ce qui concerne les protocoles de télécommunication.

Mots-clés

Systèmes temps réels - Automates temporisés - Logiques temporelles - Logiques comportementales - Langage IF - SDL - Protocoles de télécommunication - Instrumentation de code.

Une version de ce rapport ainsi que les transparents de la soutenance sont disponibles à l'adresse :

http://www.ens-lyon.fr/~ldanthon/verification/

Laboratoire Vérimag - Centre Équation - 2, avenue de Vignate 38610 GIÈRES FRANCE

Téléphone : +33 4 76 63 48 57 Télécopie : +33 4 76 63 48 50

Directeur de stage : Joseph Sifakis, Joseph.Sifakis@imag.fr

Table des matières

Introduction

La vérification de logiciels et de systèmes est un domaine récent mais en plein essor du fait du besoin constant de vérifier des systèmes critiques (leurs défauts éventuels de conception pouvant causer d'immenses dégâts tant matériels qu'humains, et donc de sérieux surcoûts financiers). Cependant, la complexification des systèmes et de leur code rend de plus en plus difficile leur validation. Pour le moment, la vérification formelle demeure trop complexe pour les systèmes dont nous disposons. Les méthodes de tests ont quant à elles des limites car elles ne peuvent être exhaustives. On fait alors appel à des méthodes de Model Checking dont la figure 1 résume le principe.

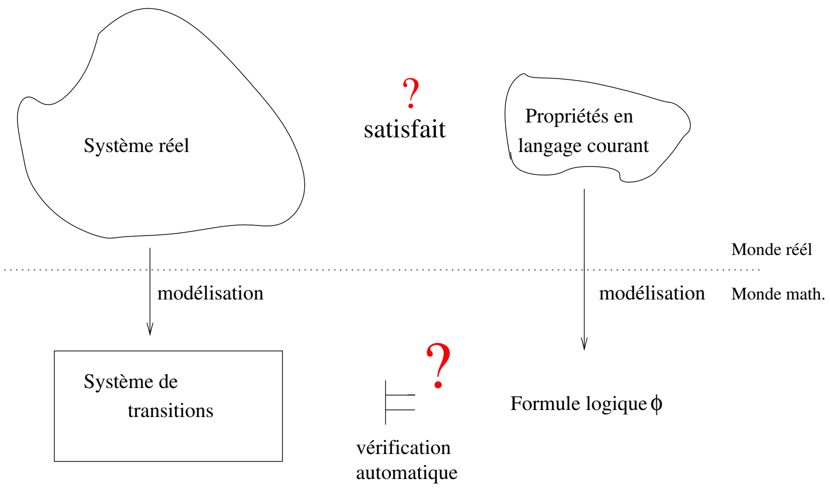

Fig. 1 – Schéma usuel de Model Checking

L'idée est de générer d'une part un modèle du système à vérifier, et d'autre part d'exprimer certaines propriétés sur ce système. Le modèle et les propriétés sont alors mises en commun pour réaliser un diagnostic, *i.e.* la phrase `true` si le modèle vérifie les spécifications requises, ou bien `false` et un contre-exemple, c'est-à-dire une exécution du système qui viole la propriété.

A l'intérieur même de ce procédé, les méthodes diffèrent sur plusieurs points :

- La transformation du protocole, du logiciel, du système à vérifier en un modèle mathématique est décrite différemment selon le but recherché.
- Les propriétés sont en général décrites à part, selon une logique adaptée.
- La vérification en elle-même est effectuée par Model Checker, lequel a été choisi en fonction du couple langage de description/type de formule à vérifier. Plusieurs méthodes ont été expérimentées : parmi elles, la méthode énumérative des états et la méthode symbolique, qui utilise des BDDs, en sont quelques unes.

Cependant, ces méthodes sont sujettes à un certain nombre de défauts, dont les principales sont la faible expressivité des formules que l'on peut vérifier, et le *state explosion problem* qui semble interdire la vérification exhaustive de systèmes réels. De plus elles semblent encore moins pratiques pour la vérification de propriétés faisant intervenir le temps.

En ce moment émerge une nouvelle technique qui est prometteuse car elle semble ne pas être soumise au *state explosion problem*. L'idée est de combiner la description du système et les exigences en un seul exécutable à vérifier. Une telle combinaison permet de ne pas générer totalement le système à vérifier, mais plutôt de vérifier "au vol" la propriété décrite sur le système. La technique va même plus loin : les exigences sont de plus en plus écrites dans le même langage que le système lui-même, modulo une différence de granularité (en fait, souvent, les exigences sont une abstraction du système réel : on désire vérifier un comportement général, plus "abstrait", que ce que l'on a décrit pour le système réel).

Parallèlement, il apparaît de plus en plus la nécessité de vérifier ce que l'on appelle des systèmes temps-réel, c'est à dire la combinaison d'un logiciel et d'un système. En effet, le temps d'exécution d'un programme, le nombre de fois qu'il exécute telle partie de son code, dépend (ou peut dépendre) de la plate-forme sur laquelle il s'exécute. Un code correct peut ainsi ne pas réaliser le comportement voulu seulement parce qu'il tourne trop lentement ou trop vite. Dès lors il s'agit de vérifier des propriétés "temporisées" de certains codes. Un modèle a été créé pour décrire ces systèmes temps-réel, ce sont les automates (ou graphes) temporisés, et des logiques temporisées pour décrire les propriétés. Cependant, plus encore que dans les cas non temporisés, la procédure classique de vérification semble trop lourde et complexe, et les propriétés sont difficiles à exprimer. En conséquence on recherche des méthodes de combinaison de la description de systèmes temporisés et de propriétés, dans une optique de simplicité pour le programmeur-vérifieur.

Le laboratoire Vérimag, dans lequel j'ai réalisé ce stage de MIM 2, a des objectifs de recherche qui vont dans ce sens. L'équipe KRONOS a développé un langage de description de systèmes temporisés, le langage IF, ainsi que de nombreuses interfaces autour de ce langage, permettant l'utilisation d'outils existants, notamment de Model Checking "classique". La recherche d'une méthode de vérification de propriétés temporisées s'est donc articulée autour de ce langage et des outils existants.

Dans ce rapport, nous établirons tout d'abord un état des lieux des méthodes existantes dans le domaine de la vérification de systèmes temporisés, nous décrirons ensuite sur un exemple les possibilités qu'offre le langage IF, notamment en ce qui concerne les protocoles de télécommunication, et ensuite nous proposerons une automatisation de la vérification de certaines propriétés comportementales temporisées.

1 Etat de l'art

Nous allons tout d'abord introduire le modèle utilisé pour décrire des sytèmes temps-réel, les automates temporisés. Nous décrirons ensuite succintement la logique TCTL , ainsi que les méthodes utilisées jusqu'à présent pour la vérification de ces systèmes. Nous conclurons par une analyse critique de ces différentes méthodes et par l'introduction de la notion d'automate observateur.

1.1 La notion d'automate temporisé

Cette section et la suivante ont été largement inspirées de la thèse [Oli94].

Tout d'abord nous allons prendre un exemple pour expliquer et introduire les concepts de base ainsi que la notation graphique des automates temporisés. Le distributeur que nous considérons permet de choisir du café ou du thé. Lorsqu'une pièce est introduite, le consommateur dispose de 10 secondes pour faire son choix. Si au bout de 10 secondes il n'a rien choisi, la pièce lui est rendue. La figure 2 décrit le graphe temporisé de ce système. Le graphe possède une seule horloge appelée x. Nous utilisons cette horloge pour mesurer le temps écoulé depuis l'introduction d'une pièce.

Le sommet 0 est le sommet initial du graphe. Il est désigné par une flèche entrante. La contrainte `true` associée à ce sommet indique qu'aucune contrainte temporelle n'est imposée, et par conséquent le système peut y rester un temps indéfini. Dès qu'une pièce est introduite dans le distributeur, l'action *pièce* est effectuée et le système franchit l'arc allant du sommet 0 vers le sommet 1. La condition `true` exprime le fait qu'une pièce peut arriver à n'importe quel moment. La valeur de l'horloge x dans le sommet 1 est égal au temps écoulé depuis l'instant où l'arc étiqueté par l'action pièce a été franchi. La condition $x \leq 10$ associée au sommet 1 indique que le système peut y rester tant que la valeur de x est inférieure ou égale à 10. Autrement dit, le système doit quitter le sommet 1 au plus tard 10 secondes après y être arrivé. Cette contrainte oblige le système à franchir un des arcs de 1 vers 0. Si $x \leq 10$ seules les transitions étiquetées par *thé* ou par *café* peuvent être franchies, étant donné que ce sont les seuls arcs qui portent la condition $x < 10$. Si x atteint la valeur 10, la seule possibilité est de franchir l'arc étiqueté *rendre*, avec la contrainte $x = 10$.

Le cycle se répète lorsque le système retourne au système 0. Ce graphe décrit donc un système qui exécute de façon cyclique une action *pièce* suivie d'une action *café* ou bien *thé* avant 10 secondes, ou bien d'une action *rendre* exactement 10 secondes après.

Cet exemple illustre comment un délai entre l'exécution de deux actions est modélisé dans un automate temporisé, que ce soit un délai strict comme dans le cas des actions *pièce* et *rendre*, ou un délai large comme dans le cas des actions *pièce* et *café*.

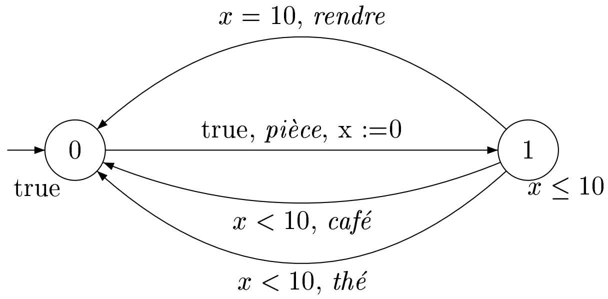

FIG. 2 – La machine à café/thé

Formalisons maintenant ces notions en donnant quelques définitions :

DÉFINITION 1.1 (HORLOGE)
Une **horloge** est une variable qui prend ses valeurs dans $\mathbb{R}^+$. Soit $X = \{x_1, x_2, \ldots, x_n\}$ un ensemble fini d'horloges. Une **valuation** des horloges est un n-uplet de $\mathbb{R}^+$, où la i-ème composante est la valeur de

l'horloge x_i. L'ensemble des valuations de X est noté $\mathbf{X}$.
La valuation $x+t$ est la valuation qui vérifie $\forall i \in [\![1,n]\!], (x+t)(i) = x(i)+t$.

DÉFINITION 1.2 (BORNE)
Une borne b est un prédicat sur l'ensemble X du type :

$$x_i\#d \text{ ou } d\#x_i \text{ ou } x_i - x_j\#c,$$

où $x_i, x_j \in X, d \in \mathbb{N}, c \in \mathbb{Z}$ et $\# \in \{<, leq\}$. On définit $\neg b$ intuitivement, par exemple $\neg(x<4) = 4 \leq x$, ...

REMARQUE 1.1 On écrit souvent $x_i \in \delta$ où δ est un intervalle sur les réels dont les extrémités sont dans $\mathbb{N} \cup \{\infty\}$ On note $\mathcal{I}$ l'ensemble de tels intervalles.

DÉFINITION 1.3 (CONTRAINTES TEMPORELLES)
L'ensemble Ψ_X des **contraintes temporelles** sur X est l'ensemble des prédicats définis de la manière suivante :

$$\psi ::= b \mid \neg\psi \mid \psi \wedge \psi,$$

où b est une borne sur X.
Pour chaque valuation x (n-uplet) et chaque contrainte $\psi \in \Psi_X$, $\psi[x]$ représente la valeur de la contrainte ψ lorsque chaque horloge x_i prend la valeur $x(i)$. On dit que la valuation x **satisfait** la contrainte ψ ssi $\psi[x]$ est vrai.
On note $[\![\psi]\!]$ pour faire référence à l'ensemble des valuations x qui satisfont ψ.

DÉFINITION 1.4 (AFFECTATION)
Une affectation α est une relation entre valuations, c'est-à-dire $\alpha \subseteq \mathbf{X} \times \mathbf{X}$. Une affectation est donnée par un ensemble $\{x_i := \delta_i \mid x_i \in X' \subseteq X \text{ et } \delta_i \in \mathcal{I}\}$

Maintenant, passons à la définition d'un graphe ou automate temporisé : dans toute la suite, E est un ensemble d'étiquettes qui est fixé.

DÉFINITION 1.5 (GRAPHE TEMPORISÉ)
Un **automate temporisé** est un quadruplet (S, X, A, I) où :
- $S = \{s_1, s_2, \ldots s_m\}$ est un ensemble fini de sommets,
- $X = \{x_1, x_2, \ldots, x_n\}$ est un ensemble fini d'horloges,
- $A \subseteq S \times E \times \Psi_X \times \mathcal{A}_X \times S$ est un ensemble finis d'arcs
- $I : S \to \Psi_X$ associe à chaque sommet une contrainte appelée *l'invariant* du sommet. On écrit souvent I_s au lieu de $I(s)$

Avant de définir formellement la sémantique des graphes temporisés, nous expliquons informelle-ment leur comportement. Soit $G = (S, X, A, I)$ un graphe temporisé :
- Un *état* du système est déterminé par un sommet $s \in S$ et par une valuation $x \in \mathbf{X}$. Nous notons un état comme une paire (s, x). Nous ne considérons que les états (s, x) pour lesquels la condition $I_s[x]$ est vraie.
- L'*arc* $(s, e, \psi_a, \alpha, s')$ peut être franchi à partir de l'état (s, x) si la contrainte ψ_a est satisfaite par la valeur des horloges. L'état résultant x' est tel que $\alpha(x, x')$
- Le franchissement d'un arc ne prend pas de temps ; autrement dit les actions sont instantanées.
- Le passage du temps ne se produit que dans les sommets.
- Les valeurs des horloges croissent uniformément avec l'écoulement du temps.
- Le système ne peut séjourner dans un sommet s tant que l'invariant I_s du sommet est satisfait par les valeurs des horloges.

Passons donc maintenant à la sémantique des graphes temporisés. On va définir la sémantique du graphe temporisé suivant les idées-clefs que l'on vient de développer, à savoir : l'écoulement d'un temps t est modélisé au moyen d'une transition étiquetée par t : ces transitions seront dites *temporelles* ; l'exécution d'une action due au franchissement d'un arc du graphe est représentée par une transition étiquetée par e et appelée *instantanée*.

DÉFINITION 1.6 (MODÈLE)
Le modèle d'un graphe temporisé (S, X, A, I) est le système de transitions étiquetés $(Q, \rightarrow)$ où :

- $Q = \{(s, x) \mid s \in S, I_s[x]\}$ est l'ensemble d'états.
- La relation de transition $\rightarrow$ est la plus petite relation $\rightarrow \subseteq Q \times (E \cup \mathbb{R}^+) \times Q$ définie par les règles suivantes :
  1. Une règle qui donne le changement d'état par écoulement d'un temps t dans le sommet s (le système reste dans s et la valeur des horloges est incrémenté du temps t) :
  $$\frac{I_s[x+t] \qquad 0 \leq t}{(s,x) \xrightarrow{t} (s, x+t)}$$
  2. Une règle qui donne le changement de (s, x) à (s', x') en franchissant l'arc $(s, e, \psi_a, \alpha, s')$:
  $$\frac{(s, e, \psi_a, \alpha, s') \in A \quad \psi_a \quad \alpha(x, x')}{(s,x) \xrightarrow{e} (s', x')}$$

Enfin, quelques notations : Soit $q = (s, x)$ un état, on note :
- $q(@)$ le sommet s.
- $q(i)$ la $i-$ème composante de x, *i.e.* x_i.
- Pour $t \in \mathbb{R}^+$, $q + t$ est l'état $(s, x + t)$.

Maintenant nous allons définir la notion de pas dans un graphe temporisé, ainsi que la notion de séquence d'exécution

DÉFINITION 1.7 (PAS)
Soient $q, q' \in Q$. On définit un **pas** de durée $t \in \mathbb{R}^+$, noté $q \rhd_t q'$, de la façon suivante :

$$q \rhd_t q' \text{ ssi } \begin{cases} q \xrightarrow{t} q' & \text{ou} \\ q \xrightarrow{t} q'' \xrightarrow{e} q' & \text{avec } q'' \in Q \text{ et } e \in E \end{cases}$$

DÉFINITION 1.8 (SÉQUENCE DE PAS)
Une **séquence de pas** est une suite infinie de pas :

$$\sigma = q_0 \rhd_{t_0} q_1 \rhd_{t_1} \dots$$

L'état q_o est appelé le *début de la séquence de pas* σ. L'ensemble des séquences de pas du modèle est noté Σ.

DÉFINITION 1.9 (POSITION)
Soit une séquence $\sigma = q_0 \rhd_{t_0} q_1 \rhd_{t_1} \dots \in \Sigma$. Une **position** π dans la séquence est une paire $(i, t) \in \mathbb{N} \times \mathbb{R}^+$ avec $t_0 \leq t \leq t_i$. L'ensemble des positions dans la séquence σ est noté $\Pi(\sigma)$.

DÉFINITION 1.10 (TEMPS ÉCOULÉ)
Le **temps écoulé** à la position (i, t) de la séquence σ est la quantité :

$$\tau_\sigma(i, t) = t + \sum_{j<i} t_j.$$

1.2 Logique TCTL

Les formules de TCTL sont construites à partir des prédicats sur les états au moyen des opérateurs logiques et de deux opérateurs temporels notés $\exists\mathcal{U}_\delta$ et $\forall\mathcal{U}_\delta$. Intuitivement, un état q d'un système temporisé satisfait $\varphi\exists\mathcal{U}_\delta\psi$ si il existe une séquence à partir de q pour laquelle la formule φ est continuellement vraie jusqu'à un état qui satisfait ψ et le temps écoulé appartient à l'intervalle δ. Cet opérateur exprime donc

une *possibilité*. De même, un état q d'un système temporisé satisfait $\varphi\forall\mathcal{U}_\delta\psi$ si pour toute séquence à partir de q, la formule φ est continuellement vraie jusqu'à un état qui satisfait ψ et le temps écoulé appartient à l'intervalle δ. On exprime alors une propriété d'*inévitabilté*.
Les formules de TCTL sont interprétées sur les états d'un système temporisé en considérant que S est l'ensemble de sommets et X l'ensemble des variables.

DÉFINITION 1.11 (ENSEMBLE CARACTÉRISTIQUE)
L'**ensemble caractéristique** d'une formule φ, noté $[\![\varphi]\!]$, est l'ensemble des états qui satisfont la formule φ.

La façon classique de vérification par Model Checking de propriétés suit alors le schéma suivant :
– on exprime les propriétés temporelles sous forme de formule TCTL .
– on calcule l'ensemble caractéristique de la formule.
– on vérifie que l'état initial appartient à cet ensemble.

Cette méthode est mise en route dans l'outil Kronos [Yov], qui a été développé au Vérimag. Cette méthode qui est totalement justifiée par la théorie a fait ses preuves dans le cas de petits systèmes, mais on atteint facilement les limites car on se ramène souvent au cas non temporisé, ce qui fait exploser le nombre d'états.

1.3 Vers une logique comportementale

Le point de vue décrit dans la section précédente paraît à certains un peu réducteur pour les raisons suivantes :
- On génère en général un modèle abstrait du code *indépendemment* des propriétés. Un tel modèle est généralement très gros en taille et long à vérifier même si certaines méthodes tendent à le faire décroître. De plus, on n'a certainement pas besoin de toutes les informations de ce modèle pour vérifier une propriété donnée.
- Les formules que l'on veut décrire sont difficiles à exprimer précisément dans les logiques temporelles dont on dispose. Par exemple, on aimerait pouvoir exprimer le comportement décrit par l'automate de la figure 3, qui signifie que chaque envoi de message est suivi par la réception du même message *sans que rien ne se soit passé entre*. Comment écrire cela selon une logique temporelle, par exemple CTL ? `AG (send_i => EF recv_i)` ? Mais cela n'exprime pas l'impossibilité d'avoir deux signaux *send* non séparés par un *recv*, donc il faut encore compliquer l'écriture. En plus du manque de lisibilité de telles formules, on n'est donc pas à l'abri d'une erreur dans leur formulation. Ici l'automate est plus pratique car il spécifie un comportement de la machine.

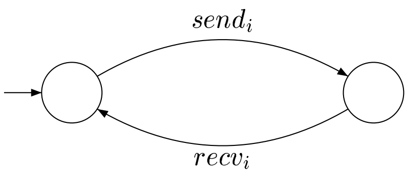

FIG. 3 – Description comportementale

En conséquence, une autre voie semble se dessiner : elle consiste en un "mélange" des propriétés du code avec le code lui même. De plus, dans le même temps, on voudrait bien s'écarter des logiques dont on dispose, qui sont des logiques déclaratives, et se diriger vers des logiques combinant des aspects déclaratifs et comportementaux. De telles logiques restent à découvrir.

L'outil MaC (Modeling and Checking) [LBA] ouvre une voie vers une nouvelle logique, plus descriptive. L'apport principal de la méthode est l'introduction de la notion d'événement (opération instantanée) et de condition (informations booléennes qui durent pendant un certain temps). Cette distinction entre événement et condition, bien que ces deux concepts soient liés, est très importante pour les vérifications de propriétés à l'exécution. L'idée est d'assigner un top d'exécution à un événement, ce qui permet aussi de mesurer des durées (entre deux événements) et donc des propriétés logiques sur les exécutions de programmes temps-réel.

La sémantique de cette logique peut être trouvée dans [LKK+99]. Notons simplement que l'on s'autorise des propriétés du type `start(run) when x=1`, *i.e.* des combinaisons événements/conditions.

1.4 Remarques sur le temps

La simplicité du modèle proposé cache en fait des difficultés inhérentes à l'introduction des aspects temporels. Nous allons ici décrire les étapes de constructions d'un modèle temporisé, ainsi que les difficultés rencontrées.

Cahier des charges
L'objectif était de modéliser des systèmes réels. En particulier, le temps ne se bloque jamais, par exemple. Dans le modèle choisi, le temps (ou sa modélisation), doit diverger. D'où un espace d'états potentiellement infini. Et la difficulté que l'on a à vérifier ces systèmes. De plus, dans les systèmes, le temps joue un rôle de synchronisation. Le temps est absolu, et en conséquence les modèles, si ils comportent plusieurs horloges, doivent faire en sorte qu'elles s'écoulent à la même vitesse.

Difficultés de vérification
Les modélisations faisant intervenir le temps sont plus difficiles à vérifier, moralement parce qu'ils font intervenir une dimension supplémentaire. Là où les systèmes ne possédaient qu'une seule dimension, la dimension "action" (avec des dépendances interactions quelquefois), les systèmes temporisés introduisent une nouvelle dimension, mais, et c'est là toute la difficulté, cette nouvelle dimension n'est pas indépendante de la première. Les actions et le temps sont interdépendants car une action peut attendre un certain temps après qu'un certain événement se soit produit, ...

En conclusion, ces difficultés sont les raisons philosophiques qui ont conduit les chercheurs à se satisfaire du modèle proposé d'une part, et qui font que la vérification de systèmes temporisés n'en est qu'à ces balbutiements.

2 Présentation de SDL et IF

Dans cette section, on présente un outil standard de description de protocoles de télécommunication, SDL, et le langage maison, IF, dont la sémantique mieux définie permet de développer des outils de vérification à comportement "garanti". Ce sont les outils que l'on a utilisé pour décrire les systèmes mis en jeu et pour vérifier des propriétés classiques sur ces systèmes.

2.1 SDL

SDL est un standard adopté en 1984 par l'Union Internationale des Télécommunications (ITU) pour décrire des protocoles de télécommunications. Il est remis à jour tout les quatre ans depuis. C'est à la fois un outil descriptif qui permet de dessiner les spécifications du protocole selon certaines règles (c'est l'outil OpenGeode) et un langage "de programmation" au sens habituel. SDL permet de décrire des systèmes communicants sous forme d'automates temporisés (avec horloges) communiquant par des canaux. Des éventuels canaux non fiables peuvent être décrits. De plus, les automates sont non déterministes ; ceci dit, à chaque branchement, le choix de la branche prise est exclusive. Plus de détails peuvent être trouvés par exemple à l'adresse `http://www.telelogic.com/products/tau/languages/sdl.cfm`.

2.2 IF

IF est un langage "maison" qui permet de décrire des systèmes temporisés communicants. La description de l'outil peut être trouvée dans [BFG+99] et quelques exemples sont développés dans [BGM01]. Dans la version 2 dont la syntaxe a été profondément remaniée, on peut même créer dynamiquement des processus, ce qui permet de décrire de nouveaux types de systèmes. Parmi les nombreux outils qui ont été autour de IF, citons notamment le traducteur SDL2IF qui permet de générer du IF à partir du SDL produit par OpenGeode. L'intérêt est qu'ensuite on dispose d'outils puissants qui permettent :

- la traduction du IF vers un système de transitions étiqueté (STE) où la notion de temps n'apparaît plus que dans les étiquettes des transitions.
- la vérification des propriétés des STE obtenus : propriétés décrites sous forme comportementale avec l'outil ALDEBARAN, et propriétés de logique temporelle avec EVALUATOR.

Cependant cette méthode de traduction vers un STE, si elle possède l'avantage de fournir une interface propre de SDL vers des outils existants, n'est pas totalement satisfaisante car en générant le STE correspondant à un système d'automates temporisés communicants, on est soumis à un *state explosion problem* : en effet, le fait de transformer des transitions gardées par une horloge en transitions simples est très coûteux en nombre d'états.

2.3 Un gros exemple avec IF

Nous allons dans cette section décrire un petit exemple en IF, qui est l'implémentation d'un protocole bien connu, le protocole BRP (Bounded Retransmission Protocol). Il y a deux processus : un émetteur et un récepteur, le premier essayant d'envoyer des messages à un second. La description que nous faisons ici est tirée de [MGK+00] et est un peu simplifiée, c'est-à-dire qu'elle se limite à des contraintes temporelles.

Le protocole essaie de réaliser le transfert d'un fichier entre deux entités : le Transmetteur et le Récepteur. Le fichier à transférer est en fait découpé en plusieurs morceaux que l'on essaie de transférer et la transmission du fichier n'est pas garanti car en cas de perte d'un paquet seul un nombre borné de retransmissions est autorisé. Dans une telle situation, les deux protagonistes doivent se mettre d'accord pour lancer le transfert du fichier suivant.

Le protocole est résumé par les graphes des figures 4 et 5.

Ces deux processus communiquent entre eux et avec leur environnement par l'intermédiaire de canaux (*buffer*) qui sont déclarés au début du programme. Le symbole `!` désigne une écriture et le symbole `?` une lecture. Une transition gardée (*i.e.* qui commence par une expression booléenne encadrée par des crochets) signifie que l'on ne peut franchir la transition que si elle est vraie. Une transition réalisant une lecture ne

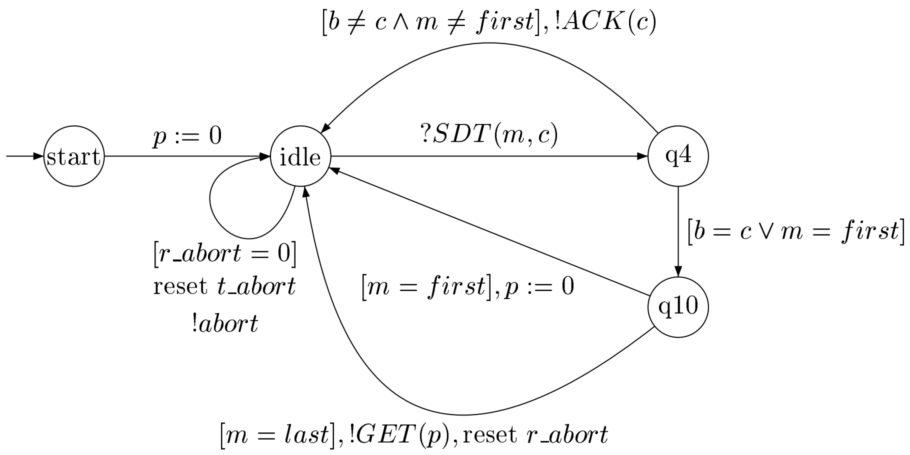

FIG. 4 – Récepteur

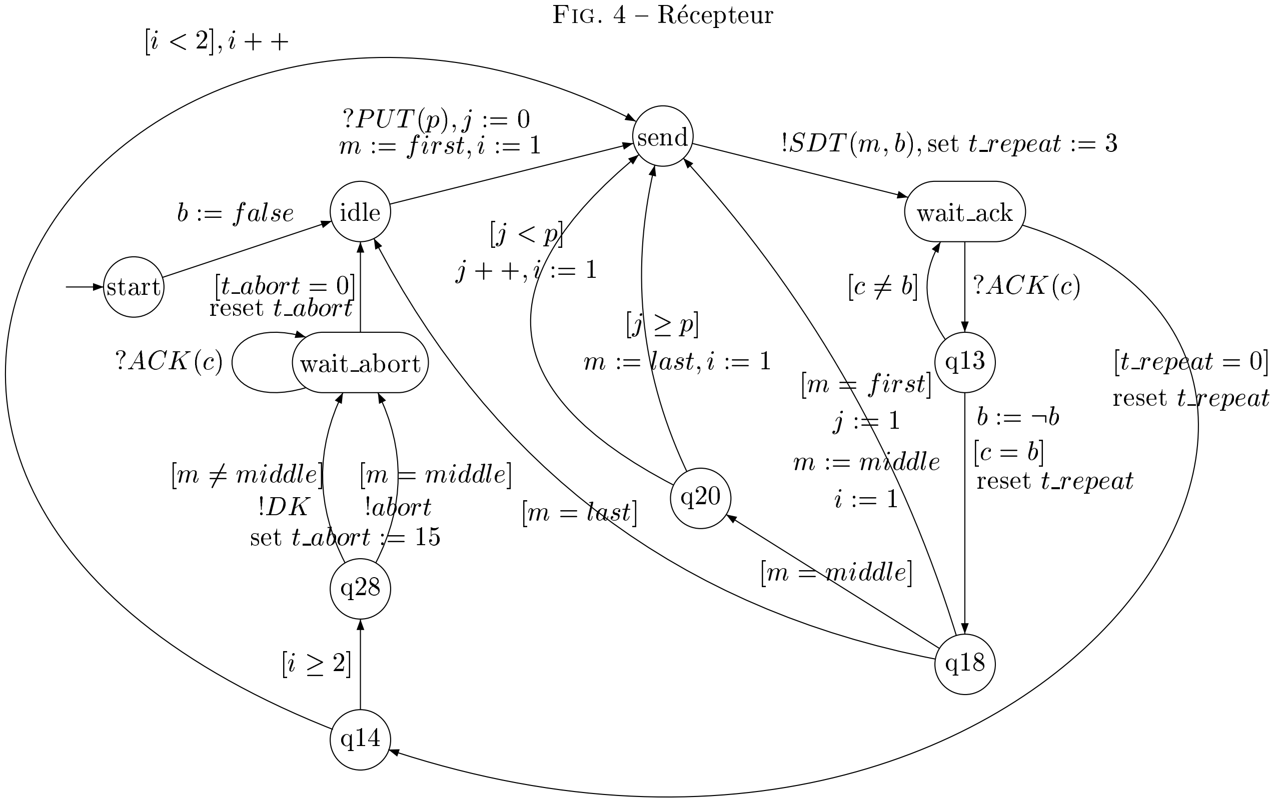

FIG. 5 – Transmetteur

peut s'effectuer que si le buffer concerné est non-vide. On voit bien que les transitions "calculent" en le sens qu'elles peuvent modifier les valeurs de certaines variables.

Le fichier à transmettre est fourni après initialisation de variables au processus Transmetteur. Le fichier est découpé en plusieurs paquets (p paquets) qu'il s'agit de transmettre à la suite, et ceci sans perte. Ce transfert de fichier est modelisé par l'envoi de la variable m qui indique si on est au début (`first`), à la fin (`last`) ou au milieu (`middle`). Pour chaque paquet, le Transmetteur attend un signal du Receiver, un `ACK`. Si un tel signal ne lui est pas transmis dans un certain délai, il conclu que le paquet s'est perdu et recommence la transmission. Si un tel scénario se repète trop (ici plus de 2 fois), le transmetteur lance un message `abort` à l'environnement, et attend que le Receiver se soit aussi aperçu du problème, et écrit un signal `abort`. Un nouveau transfert de fichier est alors relancé. Il faut quand même remarquer que le Transmetteur peut signaler qu'il ne sait pas si un message a effectivement été bien transmis, dans le cas ou il n'a pas reçu le signal `ACK` avant son timeout.

A ce stade, nous avons donc défini le système. Qu'en faire ensuite ? On dispose du programme `if2c`, qui génère des primitives spécifiques au système, primitives qui peuvent ensuite être utilisés par l'outil `generator` pour générer le STE, ou bien encore par l'outil `evaluator` avec lequel on peut vérifier directement des formules du μ-calcul.

Dans les deux paragraphes suivants, nous allons mettre en œuvre deux méthodes de vérification qui diffèrent notablement.

Première méthode

La première méthode consiste à générer le STE correspondant au système à partir du code IF de descripton, et ensuite seulement vérifier certaines propriétés temporisées (ou non !), “à vue”. C'est donc une méthode très empirique, mais qui peut tout de même servir dans le cas de petits systèmes. Nous décrivons ici sa mise en œuvre sur l'exemple du BRP.

La commande `if.open bitalt.if generator` suivie de `./generator -nt bitalt.out` permet de générer le STE, dans ce cas :

```
des(0,1308,954)
...
(2," -(transmitter_i0,env,put,{1})",4)
...
```

La première ligne signifie que cet automate a 954 états et 1308 transitions, ces transitions étant étiquetés par des informations du système IF précédent (variables, temps, signaux ...). Les transitions sont données sous la forme `(num_etat,label_trans,num_etat)`. A ce stade, cet automate contient trop d'états pour que l'on puisse y voir quelque chose. Tout d'abord, un peu d'esthétisme : l'outil Aldébaran autorise le renommage des transitions. Cela permet en plus de fusionner certains états qui se comportent identiquement, si par exemple on ne prend pas en compte les valeurs de telle ou telle variable. Par exemple, lors du traitement pour le BRP, on a seulement distingué les “put” et les “get” des différents messages possibles, en faisant :

```
rename
".*put,{1}.*"->"put(1)"
".*put,{2}.*"->"put(2)"
".*get,{1}.*"->"get(1)"
".*get,{2}.*"->"get(2)"
".*"->i
```

On peut aussi cacher certains états en faisant un fichier `.hide`, ce que l'on ne détaillera pas ici. Après ce “lifting” permis par certaines options de l'outil Aldébaran, ainsi qu'une minimisation, le système que l'on considère possède 3 états et 8 transitions. On a réalisé ce que l'on appelle une **Abstraction** et le fichier a maintenant l'allure suivante :

```
des (0, 8, 3)
(0, put(1), 1)
(0, put(2), 2)
(1, put(1), 1)
(1, put(2), 2)
(1, get(1), 0)
(2, put(1), 1)
(2, put(2), 2)
(2, get(2), 0)
```

"On voit bien" que le protocole vérifie des propriétés classiques importantes : il n'y a pas d'enchevêtrement de $put(1)$ et de $put(2)$, par exemple. . . .

Ceci dit, le processus est loin d'être automatisé. Aldébaran offre à ce stade quelques commandes intéressantes : si on dispose d'une spécification sous forme descriptive STE (par exemple obtenu à l'aide d'une description IF), l'outil peut comparer les deux automates, par exemple on peut tester si ils sont équivalents selon une certaine relation (par exemple si ils sont bisimilaires), et ainsi en déduire l'adéquation ou non des spécifications et de l'implémentation. Ceci dit, il faut quand même faire quelques renommages et/ou suppressions d'états, pour peu que le système soit gros. A titre d'exemples, le STE correspondant à une traduction IF vers STE du protocole de *token ring* pour un anneau à 4 processus possède 5000 états et 20000 transitions.

Passons maintenant à l'observation de propriétés temporisées. Le temps apparaît en effet dans les STE sous la forme de transitions `tick(i)` où i représente le nombre d'unités de temps. Le fichier `brp.aut` comporte ainsi des transitions `"tick(3)"`, qui symbolisent l'écoulement du temps pendant 3 secondes. Ceci dit, cette façon de symboliser le temps n'est pas pratique, parce que pour l'outil Aldébaran, une transition `tick(3)` n'est pas équivalente à trois transitions `tick(1)` successives. Nous avons donc écrit en Ocaml un petit parseur (128 lignes) qui permet de changer un fichier de STE en un fichier "équivalent" où toutes les transitions `tick(n)` ont été remplacées par n transitions `tick(1)` successives ; le nombre de transitions et d'états augmente donc, et il faut donc mettre à jour l'en-tête du fichier. Le fichier ainsi transformé et minimisé (en tenant compte des signaux `get, put, error, abort` et du temps) ne comporte plus que 110 états pour 189 transitions, contre 188 états et 278 transitions pour le fichier minimisé directement. En plus du gain en nombre d'états, on gagne en sens. Après cette manipulation, on devait donc pouvoir vérifier des propriétés temporelles intéressantes, incluant le temps de manière quantitative. Cela n'a pas été le cas. Les automates générés sont trop gros pour que l'on puisse "voir" les propriétés vérifiées, et l'expression/vérification s'avère trop complexe pour pouvoir être réellement utile.

Deuxième méthode La deuxième méthode consiste à exprimer des propriétés en dehors du fichier IF sous forme de formules du μ-calcul. Cette logique possède plus d'expressivité que CTL et LTL réunies, mais en général on utilise une petite partie de son pouvoir d'expression : on cherche à vérifier l'atteignabilité d'un état (en fait, on cherche à savoir si une certaine transition "mauvaise" est possible ou non, *i.e.* on cherche à vérifier des propriétés de sureté). Cette deuxième méthode est radicalement différente de la précédente principalement sur deux points :

- Il n'est pas nécessaire de générer complètement le modèle abstrait du système : la vérification de l'atteignabilité se fait *à la volée* ; et dès qu'une exécution "mauvaise" est trouvée, l'outil stoppe sa progression ;
- Les spécifications sont notées *à l'intérieur du code source IF*. C'est cette particularité que nous allons développer maintenant.

Nous allons vérifier que dans un laps de temps de 22 secondes, le transmetteur soit reçoit un signal `ACK`, soit écrit un `abort` ou un `DK`, c'est-à-dire se met dans un état d'erreur. Le but de l'exercice est donc de coder l'observateur suivant :

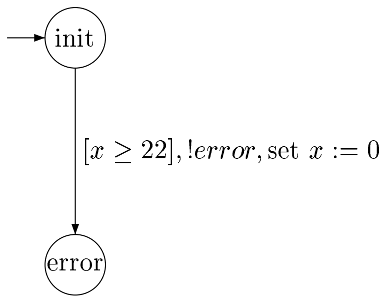

FIG. 6 – Observateur

On réalise donc le codage de cet automate dans le même fichier IF, c'est-à-dire qu'on rajoute les lignes suivantes :

```
state
```

```
  init :init;
  error;

transition
  from init if (x >= 22) do output error to env, reset x to error;
```

La nouvelle horloge que l'on a au début du programme initialisée à 0 est dans le code du protocole remise à 0 uniquement en cas de signal `ACK`, `abort` ou `DK`.

Il ne reste qu'à vérifier que l'on obtient jamais l'état d'erreur (ou, ce qui est la même chose, que l'on ne franchit jamais la transition `error`). Pour cela, on code la propriété d'atteignabilité de ladite transition dans un fichier `error.mcl` :

```
pot <" +(observer,env,error,{})"> T
```

Remarque 2.1 L'étiquette de la transition est celle donnée par Aldébaran lors de la génération du fichier `.aut`.

Maintenant, il ne reste plus qu'à vérifier que cette propriété est fausse :

```
>if.open -T brp_obs.if evaluator -v -l error.mcl
(...)
FALSE
```

Ouf ! Remarquons que cet entier 22 est charnier, si on remplace par 21 le diagonistic devient `true` et on peut éventuellement récupérer une trace d'une exécution qui mène à `error`.

En conclusion Ces divers exemples montrent toute la difficulté d'exprimer et de vérifier des propriétés selon ce shéma. Il est donc nécessaire de simplifier le processus. Ceci dit, la deuxième méthode semble plus prometteuse, car elle semble plus automatisable.

3 Propriétés usuelles et instrumentation en IF

3.1 Propriétés usuelles et cahier des charges

Lors des nombreux exemples réalisés à l'aide de If et d'Aldébaran, nous sommes arrivés à la conclusion suivante : lorsque l'on réalise des protocoles de télécommunication, certaines propriétés sont cruciales, en particulier :

- **Exclusion mutuelle** : on souhaite ici que certains processus n'accèdent pas en même temps à une "section critique", par exemple à une certaine variable partagée à un certain moment de l'exécution. Ceci n'est pas évident à réaliser et prend beaucoup de place dans l'élaboration des protocoles. Par exemple, c'est l'objet du protocole *token/ring* qui réalise cette exclusion mutuelle sur un anneau.
- **Send/Receive** : on souhaite pouvoir vérifier qu'un envoi de message est bien suivi (tout le temps ou éventuellement) d'une réception. On distingue le sens faible (éventuellement) du sens fort (tout le temps). Par exemple le protocole du bit alterné ne garantit que le sens faible. Dans le cas de systèmes temporisés, on peut parler de réponse bornée (*bounded response*), cet aspect étant important car souvent on souhaite garantir un délai de réponse.
- **No Buffer overflow** : les canaux de communication peuvent se saturer et on souhaite savoir si cela peut arriver lors de l'exécution.

Si on introduit une notion informelle d'événement, on s'aperçoit très vite que souvent, les propriétés décrites ci-dessus sont du type :

$$\forall n, d_{min} \leq occ_{n+1}(e) - occ_n(e) \leq d_{max}(1)$$

qui exprime que deux occurrences de l'événement e ne doivent jamais être rapprochés de moins de d_{min} secondes ou de plus de d_{max} secondes ; et

$$\forall n, d_{min} \leq occ_n(e_2) - occ_n(e_1) \leq d_{max}(2)$$

où e_1 et e_2 sont deux événements distincts, et qui ne doivent pas être espacés de moins de d_{min} secondes ou de plus de d_{max} secondes. C'est notemment le cas des contraintes de synchronisation, qui expriment par exemple que deux processus ne peuvent avoir accès à la même case mémoire en même temps.

On s'aperçoit dès lors de la nécessité de disposer de deux primitives :

- $occ_e : State \rightarrow \mathbb{N}$, qui à un état du système donné associe le nombre d'occurences de l'événement e qui ont déjà eu lieu.
- $date : Event \times \mathbb{N} \rightarrow \mathbb{N}$, qui associe à la n-ième occurence de e sa date d'exécution.

Ainsi, on définit $occ_n(e)$ comme raccourci :

$$occ_n(e) = d \equiv \exists s \in State, occ_e(s) = n \wedge date(e, n) = d$$

L'objectif étant de vérifier ces deux types de propriétés, deux possibilités s'offrent à nous :

- Modifier les sources d'un compilateur IF (par exemple `if.open`) afin d'avoir accès aux dates des différents événements. L'avantage est de pouvoir disposer de structures de données performantes et des événements "à la source". Le principal inconvénient est que l'on s'écarte de l'esprit d'annotation de code.
- Ecrire (ou faire écrire automatiquement) un processus `observateur` *à l'intérieur du code IF* qui code la propriété et se met dans un état d'erreur si il y a violation de cette propriété. Ensuite il n'y a plus qu'à vérifier avec les outils existants (`evaluator`) le non-atteignabilité de cet état (ou de la transition qui mène à cet état), cette formule pouvant être générée automatiquement sans trop de peine.

Nous allons étudier la faisabilité de la deuxième méthode, car travailler sur le code d'un compilateur de IF nous semble trop long et fastidieux. C'est l'objet de la prochaine section.

3.2 Mise en œuvre

Dans la suite, la notion informelle d'événement ne sera pas plus précisée, un événement pouvant être un signal émis, une condition vérifiée à un certain moment, ...

Propriété de succession d'un seul événement

La figure 7 est un automate observateur possible pour vérifier la formule (1). Dans le code du processus à vérifier, il faut en plus ajouter l'émission du signal S chaque fois que l'événement e se produit (*i.e.* à chaque occurrence de e). Et il faut coder la propriété de non atteignabilité de la transition `!error`. La petite taille de cet automate ainsi que le peu d'ajouts à l'intérieur du code IF nous font conclure que cette étape peut sans problème être automatisée.

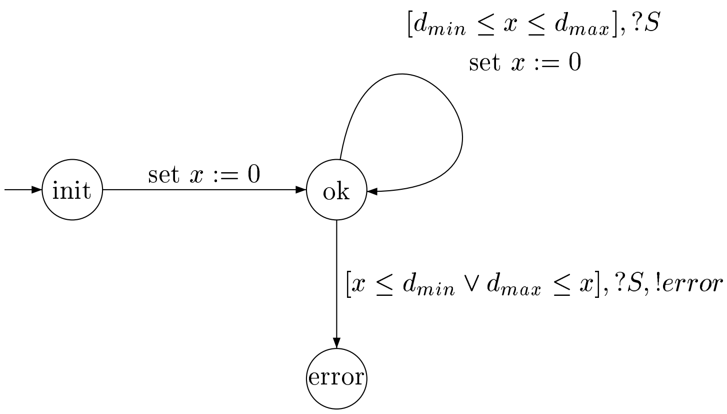

FIG. 7 – Propriété (1)

Propriété d'entrelacement de plusieurs événements

On veut vérifier des formules de type (2), en supposant que l'événement e_1 apparaît en premier (c'est par exemple la "cause" de l'événement e_2). Ce deuxième type de formule est intrinsèquement plus difficile que le précédent, car si on ne on ne suppose rien sur les événements e_1 et e_2, la mémoire requise est potentiellement infinie. En effet, si le nombre d'occurences de e_1 entre deux occurrences de e_2 est non borné, il faut une infinité d'enregistrements de l'événement e_1 entre deux occurrences de e_2. Ceci dit, si on fait des hypothèses raisonnables sur le nombre d'occurences maximum de e_1 entre deux e_2 (c'est-à-dire si on suppose pouvoir enregistrer toutes les occurrences de e_1 dans un tableau de taille raisonnable, disons de taille `MAX`, quantité que l'on nomme ***variabilité*** de e_1), alors vérifier cette propriété est possible.

• Traitons tout d'abord l'entrelacement simple : si on suppose de plus que on a alternance stricte entre les événements e_1 et e_2 alors l'automate observateur de la figure 8 convient.

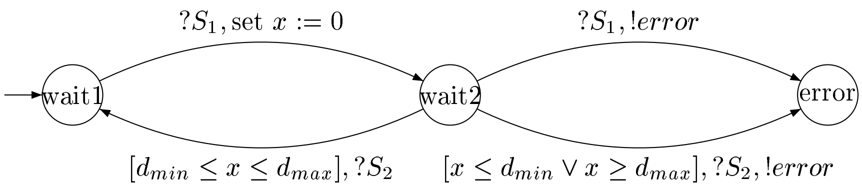

FIG. 8 – Propriété (2)-alternance simple

• Dans le cas où l'alternance stricte entre e_2 et e_1 n'est pas respectée, cela se complique. Il faut faire appel à des structures de données plus complexes et aussi à plusieurs horloges. Un premier automate observateur est celui de la figure 9. Dans cette figure, on a volontairement utilisé des primitives gérérales et des noms d'états "intuitifs" : l'idée est de pouvoir allouer une nouvelle horloge chaque fois que le signal e_1 est émis, si cette horloge dépasse d_{max}, on se place dans un état d'erreur. Si un signal e_2 "répond" à ce signal dans le délai imparti, on détruit l'horloge, sinon il y a erreur.

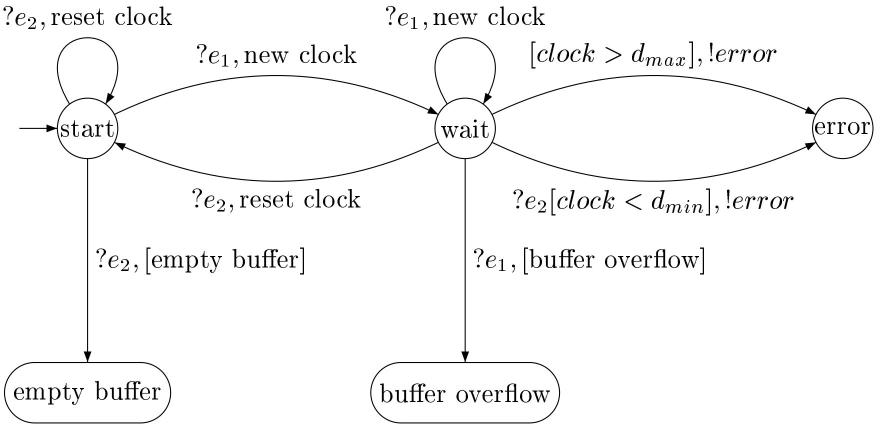

FIG. 9 – Propriété (2)-entrelacement

Lorsque l'on décide d'implémenter cet automate en IF, plusieurs difficultés se posent :

- On voit très vite la nécessité d'utiliser une structure pour stocker les dates d'arrivée des éléments. Une possibilité est de stocker un tableau d'horloges de taille `MAX`, la primitive `new clock` consistant à l'initialisation d'une horloge, la primitive `reset clock` efface la première horloge du tableau, *etc.*Mais c'est dans la gestion de ce tableau qu'apparaissent les difficultés : par exemple pour la mise à jour après la suppression d'une horloge, on ne dispose pas de boucle "for", ce qui est ennuyeux.
- La gestion du tableau d'horloges n'est pas très simple, notamment en ce qui concerne la remise à jour d'horloges.

Pour le premier problème, une solution peut être d'étaler l'automate en codant "en dur" les `MAX` horloges et en testant à chaque état les éventuels dépassements de délai . Par exemple, si `MAX` vaut 2, on obtiendrait l'automate de la figure 10, que nous avons simplifié pour faciliter la lecture. Cette solution est satisfaisante dans l'optique d'une automatisation du codage de l'automate observateur. Ceci dit, elle a l'inconvénient d'augmenter le nombre d'états de l'automate observateur, et donc la complexité de la vérification finale.

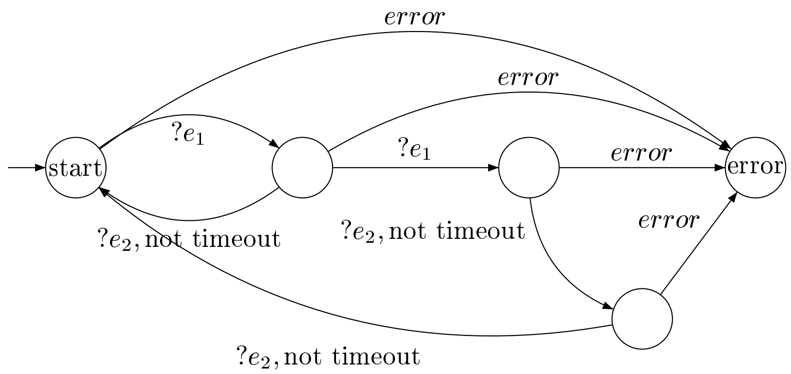

FIG. 10 – Propriété (2)- entrelacement avec MAX=2

Propriétés de jitter

Les propriétés de jitter (en anglais : instabilité) expriment la liaison de 3 occurrences d'un même événement. Si X, Y et Z sont trois dates successives d'apparition de l'événement e, alors il est quelquefois utile d'exprimer que Y se trouve "au milieu" de X et de Z, à une erreur $\varepsilon = \Delta/2$ près (voir la Figure 11).

Notons tout d'abord que cette propriété revient à la double-inégalité suivante, où l'on fait apparaître des délais :

$$(Y - X) - \Delta \leq Z - Y \leq (Y - X) + \Delta$$

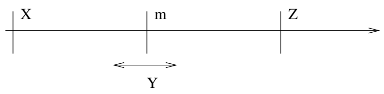

FIG. 11 – Erreur sur Y

On voit alors que l'observation de cette propriété nécessite l'emploi de deux horloges, H_1 pour la quantité $Y - X$ et H_2 pour la quantité $Z - Y$.

Un automate observateur pour cette propriété peut donc être, si l'on suit les conventions précédemment utilisées pour les notations :

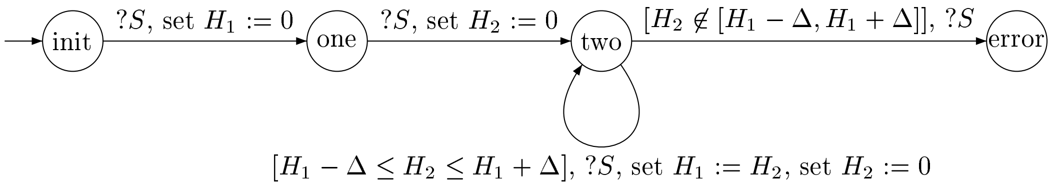

FIG. 12 – Propriété de Jitter

Ici encore, l'automatisation est tout-à-fait envisageable, et à un faible coût, vu le nombre d'horloges utilisées.

En conclusion, les problèmes rencontrés sont plutôt d'ordre méthodologique ; les difficultés de codage des automates observateurs sont bien cernées et donc disparaîtront lorsque l'on automatisera le processus. De plus, même si la taille du code IF est augmentée, cette solution est intéressante car elle ne présente pas les inconvénients du Model Checking, dans la mesure où l'on dispose d'un outil qui fait directement la vérification d'atteignabilité de l'état "erreur", à la volée. Une complète automatisation du processus est réaliste pour les deux types de propriétés données et reste à réaliser. Il reste aussi à vérifier que le nombre d'horloges utilisées ne ralentit pas trop le processus de vérification.

Conclusion et perspectives

Dans ce stage nous nous sommes familiarisés avec le domaine de pointe qu'est la vérification de systèmes temporisés. Nous avons vu que les méthodes classiques ne peuvent être utilisées avec autant d'efficacité que pour les systèmes non temporisés. Dès lors s'ouvre une nouvelle voie de recherche qui tend à mélanger implémentation et propriétés voulues, afin de minimiser le coût de la vérification.

Le travail réalisé sur le logiciel IF pourrait être poursuivi de plusieurs façons :

- Le nombre de propriétés étudiées n'est pas suffisant pour se convaincre de la faisabilité de la méthode. Un recensement le plus exhaustif possible des propriétés (temporelles ou non), ainsi qu'une étude approfondie de leur transformation en automates observateurs, reste à réaliser.
- Le codage de l'instrumentation automatique de code IF pour les propriétés étudiées pourra ensuite être envisagé. De nombreuses questions d'ordre pratique n'ont ainsi quasiment pas été discutées, notamment celle de l'accès au temps : en effet, dans certains cas, les intervalles de temps de IF semblent être bien adaptés à la gestion du temps, mais dans d'autres l'accès à une primitive `now()` qui donne la date de l'instant peut éventuellement être plus pratique. Ensuite viendra l'évaluation des performances du programme pour des propriétés concernant de gros systèmes, en comparaison avec les méthodes classiques utilisées.
- Une étude plus théorique pourrait être la suivante : quelles sont les classes de propriétés qui peuvent être vérifiées selon ce processus ? Cela pourrait déboucher sur l'invention d'une nouvelle logique, une logique comportementale, dont il faudrait définir une sémantique associée sur le langage IF. Une telle approche, plus formelle, est nécessaire si l'on veut dégager des classes de formules intéressantes pouvant être vérifiées.

Références

[BFG+99] M. Bozga, J.Cl. Fernandez, L. Ghirvu, S. Graf, J.P. Krimm, and L. Mounier. An intermediate representation and validation environment for timed asynchronous systems. In *Proceedings of FM'99 (Toulouse, France)*, September 1999.

[BGM01] Marius Bozga, Susanne Graf, and Laurent Mounier. Automated validation of distributed software using the if environment. In *Workshop on Software Model-Checking, associated with CAV'01 (Paris, France)*, July 2001.

[LBA] Insup Lee and Ben-Abdallah. A monitoring and checking framework for run-time correctness assurance.

[LKK+99] I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan. Runtime assurance based on formal specifications, 1999.

[MGK+00] M.Bozga, S. Graf, A. Kerbrat, L. Mounier, I. Ober, and D. Vincent. Sdl for real time : What is missing ? In *Proceedings of SAM'2000 (Grenoble, France)*, june 2000.

[Oli94] Alfredo Olivero. *Modélisation et Analyse de Systèmes Temporisés et Hybrides.* PhD thesis, Vérimag, 1994.

[Yov] Sergio Yovine. Kronos : A verification tool for real-time systems. (kronos user's manual release 2.2).