Spécifications de Comportements Temporisés

Matthieu Moy

Diplôme d'Études Approfondies
Informatique : Systèmes et Communcation

Année 2001 - 2002

Abstract : Timed behavior specifications

In the first part of this paper, we give a short overview of formal language theory and software verification. The next parts focus on timed regular expressions as a specification language for real-time model-checking : We define a new syntax for these expressions using *many-balanced*, or *colored* brackets, and describe the practical algorithm to compile them into timed automata. An implementation of this algorithm is also provided, and can be used as a verification tool linked to Kronos. The report ends by a short case study, and conclusion related to the usability of these expressions for software verification.

ENSIMAG — Laboratoire Verimag — UJF Grenoble

Directeurs : Oded Maler
Eugene Asarin
Jury : Nicolas Halbwachs
François Rechenmann
Christian Boitet

Table des matières

1 Contexte Scientifique **5**
1.1 La vérification des logiciels . . . 5
1.2 Rappels de théorie des langages . . . 6
1.2.1 Langages . . . 6
1.2.2 Spécification des langages . . . 6
1.2.3 Résultats importants . . . 7
1.3 Théorie des langages temporisés . . . 7
1.3.1 Introduction . . . 7
1.3.2 Définitions . . . 8
1.3.3 Automates temporisés . . . 8
1.3.4 Les expressions régulières temporisées . . . 10

2 Les expressions régulières avec parenthèses colorées **13**
2.1 Le problème . . . 13
2.1.1 Nécessité d'une notation pratique . . . 13
2.1.2 Tentative de définition de sémantique . . . 13
2.1.3 Une solution théorique . . . 14
2.2 La solution utilisée . . . 14
2.2.1 Définitions . . . 15
2.2.2 Sémantique . . . 15
2.3 Propriétés intéressantes . . . 16
2.4 Syntaxe textuelle . . . 16

3 Compilateur d'expressions régulières temporisées. **17**
3.1 Vue d'ensemble . . . 17
3.2 Structures de données . . . 18
3.2.1 La structure des expressions : Un arbre abstrait. . . . 18
3.2.2 La structure d'automate : Un graphe. . . . 18
3.3 Analyseurs lexicaux et syntaxiques . . . 18
3.4 Analyse des parenthèses colorées . . . 19
3.4.1 Principe . . . 19
3.4.2 Mise en correspondance des parenthèses . . . 19
3.4.3 Restriction de la sémantique statique . . . 19
3.4.4 Déplacement des parenthèses . . . 20
3.5 Compilation en automate . . . 22
3.5.1 Compilation des formules atomiques . . . 22
3.5.2 Opérations de base . . . 22
3.5.3 Concaténations et parenthèses . . . 24
3.6 Aspect pratique . . . 26

4 Étude de cas **28**
4.1 Introduction 28
4.2 Intégration du compilateur d'expressions dans KRONOS 28
4.2.1 Aspect théorique 28
4.2.2 Aspect pratique 28
4.3 Un exemple simple : Le passage à niveau 30
4.3.1 Description du système 30
4.3.2 Propriétés à vérifier 30
4.4 Force et faiblesses des expressions régulières temporisées. 31
4.4.1 Faiblesses 32
4.4.2 Atouts 32
4.4.3 Conclusion 33

Bibliographie **34**

A Sources Kronos des composants du passage à niveau **35**
A.1 Automate du train 35
A.2 Automate de la barrière 35
A.3 Automate du contrôleur 36

B Implementation Manual **37**
B.1 Introduction 37
B.1.1 Overview of the software 37
B.1.2 Architecture of the software 37
B.2 ASCII Regular Expressions 38
B.2.1 Basic symbols 38
B.2.2 Concatenation and color brackets 39
B.2.3 Other operators 40
B.2.4 Priority 40
B.2.5 Examples 41
B.2.6 Declaration of the alphabet 42
B.3 The Structure Of Abstract Regular Expression 42
B.3.1 Overview 42
B.3.2 The tree grammar 43
B.3.3 Decoration 43
B.4 Parsing Expressions 44
B.4.1 Context Insensitive Analysis 44
B.4.2 Context Sensitive Analysis 44
B.5 The Structure Of Automaton 45
B.6 Generic automata 45
B.7 Timed automata 45
B.7.1 Instanciation of the generic automaton 45
B.7.2 Clocks 45
B.8 Parsing Automata 46
B.9 Tests 46
B.10 Practical Aspects 46
B.11 Directory structure 46
B.12 Programming Conventions 47
B.12.1 Defensive Programming 47
B.12.2 Tracing 47
B.12.3 Input/Output 48

Introduction

La théorie des automates représente une branche importante de l'informatique. Elle a environ un demi siècle, et a trouvé des applications diverses, en particulier dans la théorie de la compilation et plus récemment dans le domaine de la vérification formelle.

A l'heure ou l'informatique trouve une des applications aussi variées et concrètes que l'automobile, l'aéronautique, les télécommunications,... la description du comportement d'un programme ne peut plus être limité à son entrée et sa sortie, mais il est nécessaire de définir précisément *quand* se produit chaque événement.

La théorie des automates temporisés est une extension de la théorie des automates, qui utilise une notion de temps continue. Une autre approche, plus récente est celle des expressions régulières temporisées. Leur équivalence aux automates est établie sur le plan théorique.

L'objet de ce projet est d'ajouter de la pratique à cette théorie en fournissant une définition des expressions dans un format le plus simple et concis possible pour l'utilisateur, et une implémentation de la transformation des expressions en automates.

Dans la section 1, nous rappellerons les bases de la vérification des logiciels et de la théorie des langages, nécessaires pour aborder en section 2 l'introduction des parenthèses colorées aux expressions régulières. Nous pourrons ensuite, en section 3, aborder les problèmes liés à la compilation de ces expressions en automates.

Enfin, la section 4 décrit l'intégration du compilateur dans l'outil Kronos, et un exemple d'utilisation de l'ensemble sur une courte étude de cas.

Chapitre 1

Contexte Scientifique

1.1 La vérification des logiciels

Dans le cycle de vie d'un logiciel traditionnel, on estime qu'environ la moitié de l'effort est consacré à la phase de tests. Malgré cette politique, les statistiques montrent que l'on peut affirmer avec peu de risques de se tromper qu'un logiciel quelconque contient toujours des erreurs.

Dans le cas ou une faute logicielle ne serait pas tolérable, pour des raisons financières ou surtout éthiques, les méthodes de tests s'avèrent donc insuffisantes. Des projets comme le métro Météore à Paris annoncent une fiabilité de 10^{-11} pannes par heures. Un simple calcul statistique montre qu'il faudrait plusieurs millions d'années de tests pour valider cet objectif.

Les développeurs de ces applications, dites « critiques » utilisent principalement les méthodes suivantes :

Tolérance aux fautes : Ces méthodes, en général basées sur la redondances, permettent au système de continuer à fonctionner en présence de fautes. Elles sont très efficaces pour les pannes matérielles, mais en général insuffisantes pour les pannes logicielles (car bien souvent, le serveur de secours à la même défaillance au même instant que le serveur primaire).

Méthodes formelles : Le principe est d'utiliser un « langage de programmation » dont la sémantique opérationnelle soit suffisamment bien définie pour être manipulée avec des outils mathématiques. La vérification est alors la mise en correspondance entre les spécifications du logiciel (souvent exprimées dans un langage dédié aux spécifications) et son implémentation.

Ce projet s'intéresse uniquement à la seconde classe de méthodes. On sait malheureusement que les problèmes informatiques ne sont pas tous décidables, et même si un problème est décidable, il peut être trop gourmand en calcul pour être résolu sur les machines réelles actuelles.

On ne peut donc pas imaginer en restant dans le modèle de la machine de Turing d'avoir un jour un outil de vérification entièrement automatique prenant en entrée les spécifications d'un logiciel et son implémentation dans des langages quelconques, et décidant si le programme est correct. Il faut donc trouver un compromis entre des spécifications dans un langage très expressif, et la décidabilité de la correction du logiciel.

Cela nous amène naturellement aux deux grandes sous-classes de méthodes formelles :

Les méthodes déductives : C'est l'approche choisie dans la méthode B, et celles basées sur les assistants de preuves comme PVS ou Coq. Elles apportent le confort d'un langage très expressif, et la satisfaction de suivre la preuve pas à pas, mais la réalisation des preuves est longue et fastidieuse, car on est en général dans un domaine indécidable.

Les méthodes algorithmiques : Dans ce cas, on perd une grande partie du pouvoir d'expression, mais les outils de vérification ont l'avantage d'être entièrement automatiques. Dans le cas ou la propriété est vérifiée, le vérificateur réponds « True », et dans le cas ou elle ne l'est pas, il donne un contre exemple.

Nous nous intéresserons tout particulièrement dans ce rapport aux méthodes algorithmiques. Ces méthodes sont en général basées sur la théorie des automates. En effet, tout programme utilisant un espace

mémoire borné[1] peut être défini par l'ensemble des valeurs de ses variables, ce qui représente un état. Son évolution est le passage d'un état à un autre en fonction des données d'entrée. Un programme apparaît donc clairement comme un ensemble d'état et un ensemble de transitions, c'est à dire un automate.

1.2 Rappels de théorie des langages

Afin de clarifier l'introduction des langages temporisés dans la partie 1.3, rappelons brièvement les bases de la théorie des langages formels.

1.2.1 Langages

Dans cette théorie, assez simplifiée, les *événements* que l'on cherche à observer sont des éléments d'un ensemble fini Σ. On ne considère pas les données numériques, qui seraient éléments d'un domaine infini et entraîneraient l'indécidabilité de la plupart des problèmes.

L'ensemble Σ est appelé *vocabulaire*, ou *alphabet*. On prend souvent les premières lettres de l'alphabet latin : $\Sigma = \{a, b, c\}$.

Un *mot*, est une séquence ordonnée d'événements. Par exemple, $abcaa$ est un mot. Un langage est un ensemble, fini ou non de mots. Par exemple, $\{a, abc, aacba\}$ et $\{a^n b^n; n \in \mathbb{N}\}$ sont des langages. Un cas particulier est le langage contenant tous les mots possibles. Il a une structure de monoïde libre, et est noté Σ^*. Tous mots sont éléments de Σ^* et tous les langages sont des sous-ensembles de Σ^*.

1.2.2 Spécification des langages

A partir du moment ou un langage est infini, il est nécessaire d'introduire un formalisme pour décrire ce langage, et pour décider de l'appartenance d'un mot au langage.

Une des classes de langages les plus simples est celle des langages réguliers. Ils sont décrits par des expressions régulières, et reconnaissables par un automate fini.

Expressions régulières

Les expressions régulières sont définies inductivement comme suit :
- $a \in \Sigma$ et ε sont des expressions régulières.
- Si φ et ψ sont des expressions régulières, alors, $\varphi \vee \psi$, $\varphi \cdot \psi$ et φ^* en sont également.

Avec la sémantique suivante :

$$\begin{array}{lcl} [\![\varepsilon]\!] & = & \{\varepsilon\} \\ [\![a]\!] & = & \{a\} \\ [\![\varphi \vee \psi]\!] & = & [\![\varphi]\!] \cup [\![\psi]\!] \\ [\![\varphi \cdot \psi]\!] & = & \{u \cdot v; u \in [\![\varphi]\!], v \in [\![\psi]\!]\} \\ [\![\varphi^*]\!] & = & \bigcup_{i=0}^{\infty} [\![\varphi]\!]^i \end{array}$$

On note souvent aussi, sans changer le pouvoir d'expression :
- $\varphi^+ = \varphi \cdot \varphi^*$
- $\varphi + \psi = \varphi \vee \psi$

Par exemple, $a \cdot (b \vee c)^* \cdot a$ est l'ensemble des mots commençant et terminant par a, et contenant n'importe quel nombre de caractères égaux à b ou c entre les deux.

Automates finis

Étant donné un langage régulier et un mot quelconque, il est relativement simple de décider de son appartenance au langage, avec un automate fini.

[1] C'est une condition nécessaire pour garantir l'exécution correcte du programme

Un automate est un quintuple (Σ, Q, I, F, T), où Σ est un alphabet, Q un ensemble d'états, $I \subset Q$ l'ensemble d'états initiaux, $F \subset Q$ l'ensemble d'états finaux, et T un ensemble de transitions (q, x, q'), avec $q, q' \in Q$ et $x \in \Sigma$.

Une trace d'exécution est une séquence $q_1, x_1, q_2, x_2, \ldots, x_{n-1}, q_n$, avec $\forall i, q_i \in Q$ et $x_i \in \Sigma$, $q_1 \in I$, $q_n \in F$, et $\forall j, (q_j, x_j, q_{j+1}) \in T$.

Un mot $m = m_1 \cdot m_2 \cdots m_n$[2] est accepté par l'automate si et seulement si il existe une trace d'exécution telle que $\forall i, m_i = x_i$.

1.2.3 Résultats importants

Les langages réguliers présentent un grand nombre de propriétés qui permettent leur traitement informatique de façon efficace :

- Le pouvoir d'expression des automates finis est le même que celui des expressions régulières (Théorème de Kleene). La transformation de l'un à l'autre est calculable.
- La classe des langages réguliers est close par union, intersection, et complémentation.[3]
- Les automates finis sont déterminisables et minimisables de manières algorithmique.

C'est ce qui explique la popularité des expressions régulières dans un grand nombre d'outils et de langages de programmation.

1.3 Théorie des langages temporisés

1.3.1 Introduction

Les résultats précédants ne sont malheureusement pas suffisants pour décrire le monde réel. En effet, il n'est pas toujours suffisant de connaître les occurrences d'événements et leur ordre, mais il peut être nécessaire de connaître les intervalles de temps qui les séparent. On peut vouloir ainsi exprimer « L'utilisateur obtient la ligne au plus tard 2 secondes après avoir décroché le téléphone », ou bien « Le train met plus de 2 secondes à arriver dans le passage à niveau après être passé devant le détecteur », pour prendre des exemples classiques.

Les langages temporisés sont aujourd'hui un domaine dynamique, et un certain nombre d'industriels s'y intéressent, souvent pour ajouter un aspect temporisé à un langage existant. Par exemple, dans [?], il est défini un formalisme graphique de spécification temps réel, et son équivalent sous forme d'automate lorsque cela est possible. Dans le même ordre d'idée, une logique extension de la logique temporelle CTL réalisée en partie par IBM est décrite dans [?].

On défini donc les langages temporisés d'une manière similaire aux langages formels non temporisés. Cette fois-ci, un mot, ou « comportement » est une succession d'événements et d'intervalles de temps. Ainsi, sur la figure 1.1, les deux comportements ne peuvent être considérés comme équivalents même si les séquences d'événements sont les mêmes.

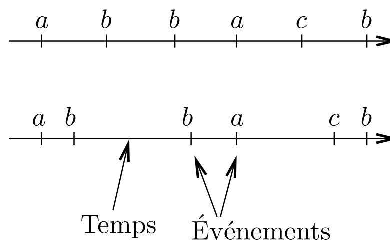

Fig. 1.1 – Langage temporisé et non temporisé

[2]En toute rigueur, $m_1 m_2$ est un mot, et $m_1 \cdot m_2$ représente l'expression « m_1 concaténé à m_2 », qui s'évalue en $m_1 m_2$. Nous confondrons les deux par abus de notation.

[3]Ce qui permet d'ajouter la règle $[\![\varphi \wedge \psi]\!] = [\![\varphi]\!] \cap [\![\psi]\!]$ sans changer le pouvoir d'expression.

1.3.2 Définitions

De la même manière que nous avons défini le monoïde libre Σ^*, on va définir le monoïde du temps.

Définition 1 (Monoïde) *Un monoïde est un ensemble muni d'une opération associative $\cdot$ et d'un élément neutre e.*

Théorème 1 *Pour tout ensemble fini Σ, Σ^*, ensemble des séquences d'éléments de Σ, est un monoïde pour la concaténation, et l'élément neutre est la séquence vide ε.*

Théorème 2 $(\mathbb{R}^+, +, 0)$ *est un monoïde.*

Dans les deux cas, l'opérateur peut être vu comme la concaténation de comportements : attendre $m+n$ secondes, c'est bien la concaténation d'une attente de m secondes avec une attente de n secondes.

Le temps et les événements étant définis, il reste à mélanger ces deux concepts : Le produit libre de deux monoïdes $(A, \diamond_a, e_a)$ et $(B, \diamond_b, e_b)$ est l'ensemble $(A \uplus B)^*$. Pour s'affranchir des représentations multiples des éléments de cet ensemble ($1+3=2+2=4$, ...), nous allons réduire cet ensemble par une relation de congruence.

Définition 2 (Relation de congruence) *Une relation de congruence $\sim$ est une relation fermée par l'opérateur du monoïde :*

$$m \sim m' \Rightarrow m_1 \cdot m \cdot m_2 \sim m_1 \cdot m' \cdot m_2$$

Définition 3 (Relation de congruence canonique) *Nous définissons la relation de congruence $\sim$ engendrée par les égalités suivantes :*

$$\begin{aligned} a_i \cdot a_j &= a_i \diamond_a a_j \\ b_i \cdot b_j &= b_i \diamond_b b_j \\ e_a = e_b &= \varepsilon \end{aligned} \tag{1.1}$$

Définition 4 (Produit libre de monoïdes) *Soient $(A, \diamond_a, e_a)$ et $(B, \diamond_b, e_b)$ deux monoïdes. Leur produit libre est $A \boxplus B = (A \uplus B)^* / \sim$, où $\sim$ est la réduction de congruence.*

Nous pouvons appliquer cette définition générique au produit de $\mathbb{R}^+$ et Σ^* :

Définition 5 (Le monoïde Temps-Événements) *Le monoïde temps-événements est $\mathcal{T}(\Sigma) = \Sigma^* \boxplus \mathbb{R}^+$.*

Un exemple d'élément de cet ensemble est

$$0.7 \cdot a \cdot b \cdot 3 \cdot 5.4 \cdot ab \cdot c \cdot 0 \cdot a \cdot \varepsilon \cdot 5.4 \cdot a \cdot 0.2$$

Qui se réduit canoniquement en :

$$0.7 \cdot ab \cdot 8.4 \cdot abca \cdot 5.4 \cdot a \cdot 0.2$$

On peut définir les projections sur les deux monoïdes composant, qui correspondent à la longueur λ, et à l'abstraction du temps μ. Dans l'exemple précédant, on a $\lambda(x) = 14.7$ et $\mu = ababcaa$.

1.3.3 Automates temporisés

Il est possible de compléter la définition des automates finis pour obtenir une classe d'automates reconnaissant ce type de langages : Les automates temporisés.

Définition

On trouve dans la littérature plusieurs définitions des automates temporisés, parfois équivalentes. Nous nous cantonnerons à celle donnée dans [**?**]. Pour éviter les ambiguïtés, nous les nommerons « automates réguliers temporisés », mais précisons que cette appellation n'a rien d'universel.

Intuitivement, on ajoute un ensemble d'horloges à l'automate. Chaque horloge a une valeur qui croit avec le temps, et qui peut être réinitialisée lors d'une transition. La valeur des horloges peut être testée par une garde sur chaque transition.

Définition 6 (Automate temporisé régulier) *Un automate temporisé régulier est un sexuplet $(Q, C, \Delta, \Sigma, s, F)$ où Q est un ensemble fini d'états, C un ensemble fini d'horloges, Σ un alphabet, Δ une relation de transition, $s \in Q$ un état initial, et $F \subset Q$ un ensemble d'états accepteurs.*

La relation de transition Δ consiste en un ensemble de quintuplés de la forme $(q, \varphi, \rho, a, q')$, où q et q' sont des états, $q \in \Sigma \cup \{\varepsilon\}$ est une lettre, $\rho \subset C$, et φ est une formule combinaison booléenne de formules de la forme $x \in I$, ou x est une horloge, et I un intervalle borné par des entiers.

La configuration d'un automate est définie par l'état dans lequel il se trouve et la valeur courante de chaque horloge.

Définition 7 (Interprétation d'horloges) *Une interprétation d'horloge est une fonction $v : C \to \mathbb{R}_+$.*

Tout ensemble $\rho \subset C$ induit une fonction $Reset_\rho$ définie par

$$Reset_\rho(v)(x) = \begin{cases} 0 & si\ x \in \rho \\ v(x) & si\ x \in \rho \end{cases}$$

Définition 8 (Configuration d'un automate) *Une configuration d'automate est une paire (q, v) où q est un état, et v une interprétation d'horloge.*

Langage accepté par un automate

Un automate peut effectuer un pas de deux façons : Il peut changer d'état, ou bien laisser le temps s'écouler.

Pas discret $(q, v) \xrightarrow{a} (q', v')$, où $a \in \Sigma \cup \{\varepsilon\}$, et il existe $\delta = (q, \varphi, \rho, a, q') \in \Delta$ tel que v satisfasse φ et $v' = \mathrm{Reset}_\rho(v)$.

Passage de temps $(q, v) \xrightarrow{t} (q, v + t\mathbf{1})$ si $t \in \mathbb{R}_+$.

Définition 9 (Exécution finie) *Une exécution finie d'un automate est une séquence de pas*

$$(q_0, v_0) \xrightarrow{z_1} (q_1, v_1) \xrightarrow{z_2} \cdots \xrightarrow{z_n} (q_n, v_n).$$

La trace d'exécution correspondante est alors la séquence $z_1 \cdot z_2 \cdots z_n$.

Définition 10 (Langage accepté par un automate) *Une exécution est dite « acceptée » si elle se termine sur un état accepteur, et que le dernier pas n'est pas un écoulement de temps*[4] [5]*. Le langage $L(A)$ défini par un automate A est l'ensemble des traces d'exécutions acceptées par cet automate.*

Exemple

Pour fixer les idées, voici un exemple simple :

Cet automate reconnaît tous les mots de durée 10 exactement, commençant par un a, suivi par une suite quelconque de b précédées d'un intervalle de temps plus court que 1. Par exemple, les mots $5.4 \cdot a \cdot 4.6 \cdot a$ et $a \cdot 0.8 \cdot b \cdot 0.7 \cdot b \cdot 0.5 \cdot b \cdot 8 \cdot a$ sont reconnus par cet automate.

Plus formellement,

$$L(A) = \{x \cdot a \cdot y_1 \cdot b \cdot y_2 \cdot b \ldots y_n \cdot b \cdot z \cdot a \mid x + z + \sum_{i=1}^{n} y_i = 10 \text{ et } \forall i \in 1..n, y_i < 1\}$$

[4]Une séquence temps-événement peut cependant se terminer par un intervalle de temps et être acceptée par un automate si la dernière transition est une ε-transition.

[5]En particulier, si $s \in F$, alors, ε est reconnu par l'automate avec l'exécution triviale $(s, 0)$

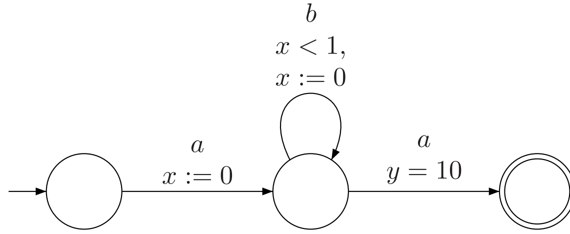

Fig. 1.2 – Exemple d'automate temporisé

Autres classes d'automates temporisés

Les particularités de cette classe d'automates par rapport à d'autres sont :

- Pas d'invariant d'état. Le comportement de l'automate n'est restreint que par les gardes des transitions.
- Les horloges croissent continuellement. Il n'est pas possible d'arrêter une horloge, ou de modifier sa vitesse.
- Les gardes des transitions sont des formules simples. Il n'est pas possible de comparer deux horloges, mais toutes la seule opération possible est de comparer une horloge à une constante.
- On travaille sur les séquences finies uniquement.
- Non déterminisme : Il est possible que deux transitions partant d'un même état portent le même symbole. Un mot est accepté si il existe un chemin accepteur.

On peut en définir d'autres, comme par exemple les automates utilisés par Kronos :

Kronos est un outil de vérification de systèmes temporisés développé à Verimag. Il prend en entrée un automate temporisé (en général donné sous la forme d'un produit de plusieurs composants) et une formule de logique temporelle, et décide de la validité de cette propriété ou de l'atteignabilité d'un état la vérifiant.

Les automates manipulés sont plus riches que les automates temporisés au sens Asarin-Caspi-Maler comme modèle de comportement :

- On autorise un invariant sur les états, qui empêche de rentrer ou de rester dans cet état si la condition n'est pas vérifiée ;
- Les gardes des transitions permettent de comparer des horloges entre elles ;
- On travaille sur des séquences infinies.

Ils décrivent cependant la même classe de langages si l'on se borne à des séquences finies.

1.3.4 Les expressions régulières temporisées

Historiquement, les premiers travaux sur les langages temporisés se sont intéressés aux automates. Cependant, de la même manière que pour les langages non temporisés, il est possible de définir un formalisme similaire aux expressions régulières, équivalent aux automates temporisés.

La définition complète de ce formalisme est donné dans [**?**]. Nous en rappelons ici les grandes lignes.

Préliminaires

Avant de donner la définition des expressions régulières, nous devons introduire deux concepts : Le renommage, et la concaténation absorbante.

Définition 11 (Renommage) *Un renommage est une fonction θ d'un alphabet dans un autre :*

$$\theta : \Sigma \rightarrow \Sigma' \cup \{\varepsilon\}$$

Définition 12 (Concaténation absorbante) *Soient $\omega \in \Sigma$ et $\omega' = x \cdot \varphi \in \Sigma$ ($x \in \mathbb{R}_+$). Si $\lambda(\omega) < x$, alors, on appelle concaténation absorbante de ω et ω' et on note $\omega \circ \omega'$ la séquence*

$$\omega \cdot (x - \lambda(\omega)) \cdot \omega'$$

Définition

Les expressions régulières sur un alphabet Σ (parfois appelées Σ-expressions) sont définies inductivement par les règles suivantes :

1. $\underline{a}$ pour toute lettre $a \in \Sigma$ et le symbole spécial ε sont des expressions.
2. Si φ, φ_1 et φ_2 sont des Σ-expressions et que I est un intervalle borné par des entiers, alors, $\langle\varphi\rangle_I$, $\varphi_1 \cdot \varphi_2$, $\varphi_1 \vee \varphi_2$, et φ^* sont des Σ-expressions.
3. Si φ, φ_1 et φ_2 sont des Σ-expressions alors, $\varphi_1 \circ \varphi_2$, $\varphi^{\circledast}$ le sont également.
4. Si φ_1 et φ_2 sont des Σ-expressions, φ_0 est une Σ_0-expression pour l'alphabet Σ_0 et si $\theta : \Sigma_0 \rightarrow \Sigma \cup \{\varepsilon\}$ est un renommage, alors, $\varphi_1 \wedge \varphi_2$ et $\theta(\varphi_0)$ sont des Σ-expressions.

Sémantique

Intuitivement, a signifie « le symbole a, sans écoulement de temps », $\underline{a}$ signifie « le symbole a, précédé d'un intervalle de temps quelconque », et ε désigne la chaîne vide. Les parenthèses $\langle\varphi\rangle_I$ représentent une contrainte de temps : La longueur de l'expression φ doit être incluse dans I. « $\cdot$ » est l'opérateur de concaténation, et φ^* signifie, comme pour les expressions régulières classiques, « φ, répété un nombre indéterminé de fois ». Les opérateurs $\vee$ et $\wedge$ correspondent respectivement à l'union et à l'intersection de langages.

Notons que la concaténation absorbante n'est pas strictement nécessaire et n'augmente pas le pouvoir d'expression puisqu'il existe un algorithme pour l'éliminer.

Formellement, la sémantique est définie comme suit :

$$
\begin{array}{lcl}
[\![\varepsilon]\!] & = & \{\varepsilon\} \\
[\![\underline{a}]\!] & = & \{r \cdot a : r \in \mathbb{R}_+\} \\
[\![\langle\varphi\rangle_I]\!] & = & [\![\varphi]\!] \cap \{u : \lambda(u) \in I\} \\
[\![\varphi_1 \vee \varphi_2]\!] & = & [\![\varphi_1]\!] \cup [\![\varphi_2]\!] \\
[\![\varphi_1 \cdot \varphi_2]\!] & = & [\![\varphi_1]\!] \cdot [\![\varphi_2]\!] \\
[\![\varphi^*]\!] & = & \bigcup_{i=0}^{\infty}([\![\underbrace{\varphi \cdot \cdots \cdot \varphi}_{i \text{ fois}}]\!])
\end{array}
$$

$$
\begin{array}{lcl}
[\![\varphi_1 \circ \varphi_2]\!] & = & [\![\varphi_1]\!] \circ [\![\varphi_2]\!] \\
[\![\varphi^{\circledast}]\!] & = & \bigcup_{i=0}^{\infty}([\![\underbrace{\varphi \circ \cdots \circ \varphi}_{i \text{ fois}}]\!])
\end{array}
$$

$$
\begin{array}{lcl}
[\![\varphi_1 \wedge \varphi_2]\!] & = & [\![\varphi_1]\!] \cap [\![\varphi_2]\!] \\
[\![\theta(\varphi)]\!] & = & \{\theta(u) : u \in [\![\varphi]\!]\}
\end{array}
$$

On y ajoute en général les notations :

$$
a = \langle\underline{a}\rangle_0; \quad \varphi^+ = \varphi \cdot \varphi^*; \quad \varphi^{\oplus} = \varphi \circ \varphi^{\circledast}; \quad \varphi^{\circ i} = \underbrace{\varphi \circ \cdots \circ \varphi}_{i \text{ fois}}
$$

Utilité du renommage et de l'intersection

Nous avons vu plus haut que l'intersection n'apportait rien au pouvoir d'expression des expressions régulières non temporisées, et il en est de même pour le renommage.

Intersection Dans le cas des langages temporisés, la situation est différente, puisqu'un automate peut avoir plusieurs horloges, et donc imposer plusieurs contraintes de temps indépendantes sur la même séquence. Un exemple classique est :

$$\langle \underline{a} \cdot \underline{b} \rangle_3 \cdot \underline{c} \wedge \underline{a} \cdot \langle \underline{b} \cdot \underline{c} \rangle_3$$

Qui est reconnu par l'automate de la figure 1.4, mais pas exprimable sans intersection.

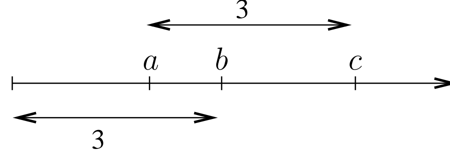

Fig. 1.3 – Contraintes de temps croisées

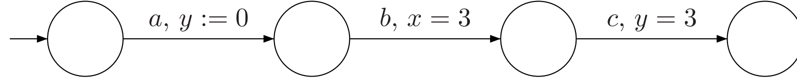

Fig. 1.4 – Automate reconnaissant le langage

Renommage Ceci n'est toujours pas suffisant. Le cas le plus simple nécessitant un renommage est celui des séquences se terminant par un intervalle de temps. On peut par contre les définir comme l'image de $w \cdot \langle \underline{a} \rangle_r$ en associant a à ε. Pour ces cas, on pourrait définir une syntaxe intermédiaire, autorisant une expression τ de sémantique $[\![\tau]\!] = \mathbb{R}$, pour représenter les intervalles de temps explicitement. Néanmoins, cette solution n'est toujours pas suffisante pour exprimer des langages comme

$$\{r_1 \cdot a \cdots r_k \cdot a : 1 < j < k \text{ et } \sum_{i=1}^{j} r_i = \sum_{i=j}^{k} r_i = 1\}$$

Ce langage peut être défini par l'image de l'expression

$$\langle \underline{a}^+ \cdot \underline{b} \rangle_1 \cdot \underline{a}^+ \wedge \underline{a}^+ \cdot \langle \underline{b} \cdot \underline{a}^+ \rangle_1$$

par le renommage $\theta : a \mapsto a, b \mapsto a$. Il est prouvé dans [**?**] qu'il n'est pas possible de l'exprimer sans renommage.

Résultat important : Le théorème de Kleene temporisé

Un résultat important sur les expressions régulières temporisé est leur équivalence avec les automates réguliers temporisé. Nous nous intéresserons ici à la transformation des expressions en automates. Un algorithme de transformation théorique est donné dans [**?**]. Nous étudierons en détails un algorithme pratique pour transformer les expressions efficacement dans la partie 3. Le théorème peut s'énoncer de la manière suivante :

Théorème 3 (« Kleene ») *Soit L un langage temporisé sur Σ. Alors,*

$$\exists A \mid L = L(A) \iff \exists E \mid L = L(E)$$

où A désigne un automate, et E une Σ-expression.

C'est ce résultat qui permet d'utiliser les expressions régulières comme langage de spécification pour des systèmes que l'on souhaite vérifier à l'aide d'automates.

Chapitre 2

Les expressions régulières avec parenthèses colorées

2.1 Le problème

2.1.1 Nécessité d'une notation pratique

La définition des expressions régulières telle que nous l'avons donnée dans la section 1.3.4 fait intervenir le renommage et l'intersection, ce qui est tout à fait souhaitable sur le plan théorique, pour conserver un pouvoir d'expression équivalent aux automates, mais est gênant sur le plan pratique. En effet, un utilisateur de ces expressions n'aura sûrement pas envie de recopier l'expression plusieurs fois pour exprimer plusieurs contraintes de temps. Dans l'exemple d'utilisation du renommage et de l'intersection de la section 1.3.4, on peut dire que l'expression donnée n'a rien d'intuitive.

Une solution serait d'introduire directement la notion d'horloge dans les expressions, comme cela est fait dans [?]. L'expression $(\langle \underline{a} \cdot \underline{b} \rangle_3 \cdot \underline{c}) \wedge (\underline{a} \cdot \langle \underline{b} \cdot \underline{c} \rangle_3)$ s'écrirait alors

$$(a, x_2 := 0) \cdot (x_1 = 3, b) \cdot (x_2 = 3, c)$$

Cette solution n'est pas retenue, car le but est de créer un langage de spécifications différent des automates, donc, d'éviter l'utilisation explicite des horloges dans les expressions. Elle permet cependant de se passer du renommage et de l'intersection.

La seconde solution est d'utiliser des parenthèses mal imbriquées, ou « parenthèses colorées ». L'expression précédante s'écrivant alors

$$\langle \underline{a} \cdot \lfloor \underline{b} \rangle_3 \cdot \underline{c} \rfloor_3$$

Intuitivement, il faut lire ces deux paires de parenthèses comme deux contraintes de temps indépendantes.

2.1.2 Tentative de définition de sémantique

Le problème est que ces parenthèses colorées cassent la définition inductive des expressions régulières temporisées. Il faut donc trouver une autre définition qui permette de leur donner une syntaxe bien définie et une sémantique claire.

La définition inductive permettait une structure en arbre correspondant à l'arbre de dérivation de la grammaire des expressions (en assimilant la définition inductive à une grammaire hors contexte). Dans ce cas, les parenthèses expriment à la fois une contrainte de temps et une modification de la priorité dans les opérations. Par exemple, l'expression $\underline{a} \cdot \langle \underline{b} + \underline{c} \rangle_1$ est équivalente à $\underline{a} \cdot \langle (\underline{b} + \underline{c}) \rangle_1$ et non à $(\underline{a} \cdot \langle \underline{b}) + \underline{c} \rangle_1$, qui d'ailleurs n'aurait aucun sens.

En voulant représenter des expressions aussi simples que $\langle \underline{a} \cdot \lfloor \underline{b} \rangle_3 \cdot \underline{c} \rfloor_3$, il devient difficile de dire quel est l'opérateur de plus haut niveau de priorité. La première idée est d'appliquer un « filtre de couleur », pour chaque couleur de parenthèse. On applique à l'expression un renommage qui associe à chaque couleur

de parenthèses sauf une, la chaîne vide. L'expression obtenue ne comporte alors qu'une couleur, et doit par conséquents être bien parenthésée. Dans notre exemple, on obtient $\underline{a} \cdot \lfloor \underline{b} \cdot \underline{c} \rfloor_3$ et $\langle \underline{a} \cdot \underline{b} \rangle_3 \cdot \underline{c}$. Il faut alors combiner ces expressions.

Le problème est que même si leur écriture est similaire, leur structure est en réalité différente, comme l'illustre la figure 2.1

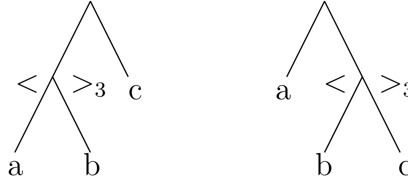

Fig. 2.1 – Structure des expressions obtenues par filtre de couleur

On peut simplement prendre l'intersection des langages, mais cette solution présente deux inconvénients : Elle fait intervenir le produit cartésien au niveau des automates, ce qui est une opération coûteuse, et ne règle pas le problème du renommage.

En effet, rappelons l'exemple de la section 1.3.4. L'expression avec renommage est $(\langle \underline{a}^+ \cdot \underline{b} \rangle_1 \cdot \underline{a}^+) \wedge (\underline{a}^+ \cdot \langle \underline{b} \cdot \underline{a}^+ \rangle_1)$ par le renommage $\theta : a \mapsto a, b \mapsto a$. Il semblerait naturel de l'exprimer de la manière suivante en utilisant les parenthèses colorées :

$$\langle \underline{a}^+ \cdot \lfloor \underline{a} \rangle_3 \cdot \underline{a}^+ \rfloor_3$$

Or, si on applique le filtre de couleur, et que l'on prends l'intersection des expressions obtenues, le résultat est $(\underline{a}^+ \cdot \lfloor \underline{a} \cdot \underline{a}^+ \rfloor_3) \wedge (\langle \underline{a}^+ \cdot \underline{a} \rangle_3 \cdot \underline{a}^+)$ et le langage défini contient, par exemple, l'expression $(3 \cdot a \cdot a) \cdot 100 \cdot (3 \cdot a \cdot a)$, ce qui n'est pas du tout ce que l'on cherche à faire.

2.1.3 Une solution théorique

Une solution satisfaisante sur le plan théorique est développée dans [**?**] en donnant une sémantique en deux étapes aux expressions : Dans un premier temps, on considère l'expression comme une expression régulière non temporisée, les parenthèses étant des symboles comme les autres. Cette première étape définie un langage dont les éléments doivent être des expressions régulières temporisées bien parenthésées. La seconde étape est de définir le langage correspondant à chacune des expressions générées.

Cette solution à le gros avantage d'être complète, sans utiliser ni renommage ni intersection. Elle autorise l'écriture d'expressions au premier abord troublantes, comme :

$$\langle \underline{a} \cdot \langle \underline{b} \rangle_I \cdot \langle \underline{c} \rangle^* \cdot \underline{d} \rangle_J$$

qui se développe en

$$\langle \underline{a} \cdot \underline{d} \rangle_J, \qquad \langle \underline{a} \cdot \underline{b} \rangle_I \cdot \langle \underline{c} \cdot \underline{d} \rangle_J, \qquad \langle \underline{a} \cdot \underline{b} \rangle_I \cdot \langle \underline{c} \cdot \underline{b} \rangle_I \cdot \langle \underline{c} \cdot \underline{d} \rangle_J, \quad \ldots$$

Le défaut de cette solution est qu'elle autorise également l'écriture de formules très difficiles à interpréter intuitivement, comme

$$\langle \underline{a} \cdot (\langle \underline{b} \rangle_I \cdot \langle \underline{c} \rangle \vee \underline{e})^* \cdot \underline{d} \rangle_J$$

Un langage de spécification issu de ce formalisme sera donc difficile à utiliser, et « dangereux », au sens où il autorisera l'utilisateur à exprimer des propriétés qui ne correspondent pas forcement à l'interprétation qu'il en attend.

2.2 La solution utilisée

La solution présentée ici n'a pas la prétention d'être universelle, ou meilleure que les autres dans l'absolu. Elle paraît satisfaisante dans l'optique d'une utilisation pratique, du fait qu'elle permet un traitement algorithmique efficace, et reste relativement simple à utiliser.

La philosophie est de n'autoriser les parenthèses que lorsque l'on peut leur donner une sémantique claire et intuitive. Elles ne suffisent pas à éliminer complètement l'intersection et le renommage, mais doivent être considérées comme un raccourcis syntaxique

Le premier constat est qu'il n'est logique de donner du sens qu'à des paires de parenthèses se trouvant dans la même liste de concaténation. Comment interpréterions-nous par exemple l'expression $\langle \underline{a} + \lfloor \underline{b} \rangle_3 + \underline{c} \rfloor_3$?

Nous redéfinissons donc l'opérateur de concaténation non plus comme un opérateur binaire, mais comme un opérateur n-aire. Chaque élément d'une liste de concaténation étant une expression à laquelle sont ajoutées les parenthèses ouvrantes et fermantes qui l'entourent.

Il faut donc désormais voir l'expression

$$\langle \underline{a} \quad \cdot \quad \lfloor \underline{b} \rangle_3 \quad \cdot \quad \underline{c} \rfloor_3$$

comme une écriture abrégée de

$$\Big(\quad (\{\langle\}, a, \emptyset) \quad \cdot \quad (\{\lfloor\}, \underline{b}, \{\rangle_3\}) \quad \cdot \quad (\emptyset, \underline{c}, \{\rfloor_3\}) \quad \Big)$$

La règle de validité est que toute parenthèse ouvrante est fermée avant la fin de l'expression, et toute parenthèse fermante est ouverte avant d'être fermée.

2.2.1 Définitions

La nouvelle définition des expression peut être donnée comme suit :

Définition 13 (Atome parenthésé) *Un atome parenthésé est un triplet (O, E, C), où O est un ensemble de parenthèses ouvrantes, E une expression, et C un ensemble de parenthèses fermantes. On note $\mathcal{O}(O, E, C) = O$, $\mathcal{E}(O, E, C) = E$ et $\mathcal{C}(O, E, C) = C$. Pour une parenthèse fermante $\mathcal{P}$, on note* opening(P) *la parenthèse ouvrante correspondante, et pour une parenthèse ouvrante $\mathcal{O}$,* closing(O) *la parenthèse fermante correspondante.*

Par exemple, $A = (\{\langle, \lfloor\}, a, \{\rfloor\})$ est un atome parenthésé, avec $\mathcal{O}(A) = \{\langle, \lfloor\}$ et $\mathcal{E}(A) = \{\rfloor\}$.

Dans $\langle \underline{a} \cdot \lfloor \underline{b} \rangle_3 \cdot \underline{c} \rfloor_3$, on a opening$(\rfloor_3) = \lfloor$ et closing$(\lfloor) = \rfloor_3$.

Définition 14 (Expression régulière temporisées avec parenthèses colorées) *Les expressions régulières temporisées avec parenthèses colorées sont définies inductivement avec les règles suivantes :*

1. *$\underline{a}$ pour toute lettre $a \in \Sigma$ et le symbole spécial ε sont des expressions.*
2. *Si φ, φ_1 et φ_2 sont des Σ-expressions et que I est un intervalle borné par des entiers, alors, $\langle \varphi \rangle_I$, $\varphi_1 \vee \varphi_2$, et φ^* sont des Σ-expressions.*
3. *Si φ, φ_1 et φ_2 sont des Σ-expressions alors, $\varphi_1 \circ \varphi_2$, $\varphi^{\circledast}$ le sont également.*
4. *Si φ_1 et φ_2 sont des Σ-expressions, φ_0 est une Σ_0-expression pour l'alphabet Σ_0 et si $\theta : \Sigma_0 \to \Sigma \cup \{\varepsilon\}$ est un renommage, alors, $\varphi_1 \wedge \varphi_2$ et $\theta(\varphi_0)$ sont des Σ-expressions.*
5. *Soient $e_1, e_2, \ldots, e_n$ des atomes parenthésés. $(e_1 \cdot e_2 \cdot \cdots \cdot e_n)$ est une expression si $\forall i \in 1..n, \forall O \in \mathcal{O}(e_i), \exists j \in i..n \mid \text{closing}(O) \in \mathcal{C}(e_j)$ et $\forall i \in 1..n, \forall C \in \mathcal{C}(e_i), \exists j \in 1..i \mid \text{opening}(C) \in \mathcal{O}(e_j)$. Dans le cas où l'expression contient plusieurs paires de parenthèses identiques, la correspondance est faite de la manière habituelle.*

2.2.2 Sémantique

Nous pourrions définir la sémantique de ces expressions relativement aisément à l'aide d'automates, mais il semble préférable de le faire de manière indépendante. Nous donnerons l'algorithme de transformation dans un second temps.

On défini alors la sémantique de E inductivement avec les mêmes règles que pour les parenthèses bien balancées, plus la suivante :

– Pour une liste de concaténation

$$E = \Big(\quad (O_1, E_1, C_1) \quad \cdot \quad (O_2, E_2, C_2) \quad \cdot \quad \cdots \quad \cdot \quad (O_n, E_n, C_n) \quad \Big)$$

la sémantique de E est donnée par

$$\begin{aligned} [\![E]\!] = \big\{ e_1 \cdot e_2 \cdot \ \cdots \ \cdot e_n \quad & | \quad \forall i \in 1..n, e_i \in [\![E_i]\!] \\ & \text{et} \quad \forall i \in 1..n, \forall j \in 1..n, \forall o \in \mathcal{O}(e_i), \forall c_I \in \mathcal{C}(e_j), \\ & \qquad o = \text{opening}(c_I) \Rightarrow \sum_{k=i}^{j} \lambda(e_i) \in I \big\} \end{aligned}$$

En d'autres termes, on concatène les expressions issues de la sémantique des composants de la liste, comme on l'aurait fait avec une concaténation classique sans parenthèses, mais on en prends la restriction aux éléments qui vérifient les contraintes de temps.

2.3 Propriétés intéressantes

Le pouvoir d'expression de cette classe d'expression est le même que pour les expressions régulières temporisées « classiques », et sont également équivalentes aux automates temporisés.

Théorème 4 *Soit L un langage temporisé sur Σ. Alors,*

$$\exists E \mid L = L(E) \iff \exists A \mid L = L(A)$$

$$\exists E' \mid L = L(E') \implies \exists A \mid L = L(A)$$

où A désigne un automate, et E une expression régulière temporisée avec parenthèses colorées, et E' une expression régulière temporisée sans renommage ni intersection.

Elles permettent *dans la plupart des cas* de se passer du renommage et de l'intersection. Nous conjecturons le fait que ceux-ci restent nécessaires, comme par exemple dans l'exemple suivant :

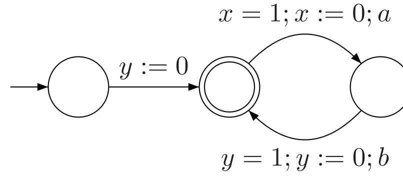

Fig. 2.2 – Exemple d'automate nécéssitant l'intersection.

Le langage reconnu est exprimable sans problème avec intersection : $(\langle \underline{a} \cdot \underline{b} \rangle_1)^* \wedge (\langle \underline{a} \rangle_1 \cdot (\langle \underline{b} \cdot \underline{a} \rangle_1)^* \cdot \underline{b})$, mais il ne semble pas possible de l'exprimer sans.

Pour des spécifications d'une complexité raisonnable, l'utilisateur ne devrait pas avoir besoin de ces opérateurs.

2.4 Syntaxe textuelle

Nous avons jusqu'ici ignoré les problèmes concrets de notations. Pour écrire un compilateur, il est néanmoins nécessaire de définir une syntaxe textuelle pour ce langage.

La grammaire complète est donnée en annexe B.2

Chapitre 3

Compilateur d'expressions régulières temporisées.

3.1 Vue d'ensemble

Le but du logiciel développé est de transformer une expression régulière temporisée en automate temporisé au format KRONOS.

La philosophie est de faire un outil le plus générique possible pour permettre une réutilisation du code pour des projets similaires. C'est dans cet esprit que les aspects syntaxiques ont été isolés dans des modules séparés, et que la transformation proprement dite se fait exclusivement sur des structures abstraites.

Le présent document n'a pas vocation d'être une documentation technique et insiste surtout sur les aspects théoriques. Pour plus de détails techniques, se référer à la documentation d'implémentation.

L'architecture du projet est donc la suivante :

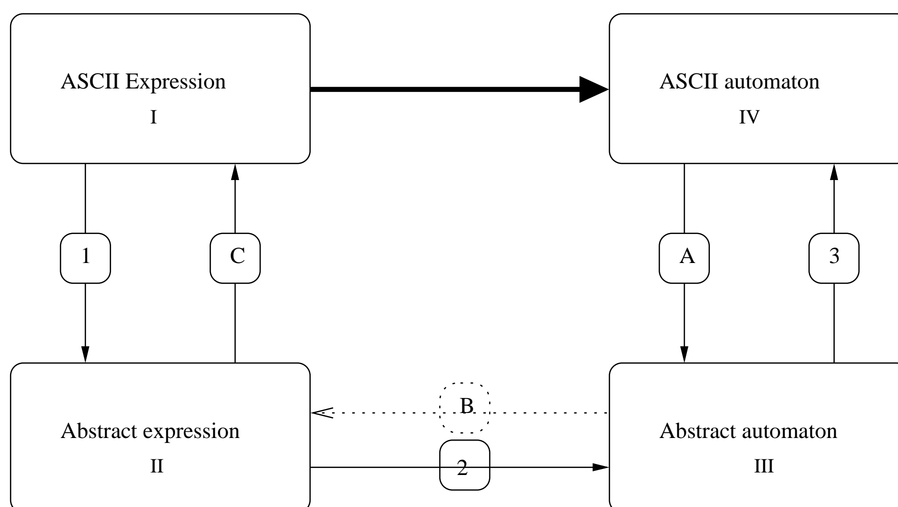

FIG. 3.1 – Architecture du compilateur.

(L'étape B n'est pas encore implémentée.)

La flèche en gras correspond à la transformation visible par l'utilisateur, qui est en fait décomposée en 3 passes :

- Analyse lexicale et syntaxique de l'expression et génération d'une structure abstraite ;
- Transformation de l'expression abstraite en automate abstrait ;
- Génération du fichier correspondant en syntaxe KRONOS.

La partie algorithmique est donc écrite une fois pour toutes, et il est possible de changer la syntaxe d'entrée ou de sortie de manière relativement aisée. C'est ainsi qu'une sortie au format BCG a pu être implémentée en 97 lignes de code seulement. (Ceci permet d'utiliser l'outil de visualisation d'automate BCG_EDIT, qui fait partie de l'ensemble CADP).

Il nous faut donc définir 2 formats textes et 2 structures abstraites.

3.2 Structures de données

3.2.1 La structure des expressions : Un arbre abstrait.

Les expressions sont représentées par une structure d'arbre. Un arbre est un pointeur sur un noeud, qui contient de l'information et des pointeurs vers ses fils. Le nombre de fils par noeud est variable.

Chaque constructeur implémente des vérifications sur la nature et le nombre de fils de chaque noeud, si bien qu'on a la garantie à tout moment que l'arbre vérifie la grammaire qui le définie.

3.2.2 La structure d'automate : Un graphe.

Toujours dans un esprit de réutilisabilité du code, la structure d'automate est définie en deux temps : Un automate générique, dont hérite un automate temporisé.

Un automate est vu comme un ensemble d'états, portant éventuellement de l'information dans une étiquette, reliés par des transitions, portant également une étiquette. L'automate est un paquetage générique paramétré par le type des étiquettes qui marqueront les transitions et les états. Il défini les types automates, transition, et états, et les primitives pour les manipuler.

L'automate temporisé est défini à partir d'une instanciation de ce paquetage avec une contrainte de temps et un symbole comme étiquette sur les transitions. Le type est alors un type dérivé de cette instanciation, ce qui permet de redéfinir certaines opérations, et d'ajouter des champs aux différents types définis dans le paquetage générique. Les opérations réutilisent au maximum les primitives de l'automate générique. Par exemple, pour le produit cartésien, l'automate temporisé ne gère que les intersections de contraintes de temps, mais c'est l'automate générique qui crée les états et les transitions, et calcule l'atteignabilité.

Il est ainsi plus facile de créer une variante d'automate basée sur le même patron, en gardant un code factorisé au maximum.

3.3 Analyseurs lexicaux et syntaxiques

Le langage des parenthèses colorées est un langage sous contexte. En effet, on sais que $\{a^n \cdot b^m \cdot c^n \cdot d^m \mid n, m \in \mathbb{N}\}$ n'est pas hors contexte, et cela représente un des cas d'imbrication de parenthèses colorées.

De plus, on ne borne pas *a priori* le nombre de couleurs de parenthèses. On travaille donc sur un vocabulaire infini. C'est un cas similaire au problème des déclarations de variables en théorie de la compilation classique.

La solution est de faire l'analyse en deux passes. La première ne différencie pas les différentes couleurs, et ne considère que les éléments lexicaux « parenthèse ouvrante » et « parenthèse fermante » (en plus bien sûr de tous les éléments autres que les parenthèses).

Lors de cette passe, les parenthèses seront attachées au symbole ou à l'expression à laquelle elle se rapproche, pour former un atome parenthésé.

Cette partie étant relativement classique sur le plan théorique, elle ne sera pas détaillée ici. Un analyseur lexical découpe le flot d'entrée au format texte en un flot de éléments lexicaux (lexèmes). Un analyseur syntaxique permet alors de prendre en compte la syntaxe proprement dite.

La grammaire détaillée des expressions régulières au format texte est donnée en annexe B.2.

On obtient alors un arbre abstrait, que l'on parcourt une seconde fois pour faire la mise en correspondance des parenthèses.

3.4 Analyse des parenthèses colorées

3.4.1 Principe

La seconde passe va réaliser 3 opérations sur l'expression :

- Transformation des concaténation binaires en listes de concaténation.
- Mise en correspondances des parenthèses
- Modification de la structure de l'expression lorsque cela est possible pour optimiser celle-ci.

L'algorithme consiste en un parcours infixé de l'arbre (C'est à dire un parcours dans le sens de lecture pour l'expression au format texte). On ne change rien sur les noeuds correspondant aux opérateurs autres que les concaténations.

Lorsqu'on rencontre un noeud « concaténation », on passe dans un mode de transformation de concaténation, et on y reste tant qu'on ne trouve pas un opérateur différent du premier. (Attention, on ne mélange pas la concaténation standard « $\cdot$ » et la concaténation absorbante « $\circ$ » qui ne sont pas associatives). Il est bien entendu possible d'avoir plusieurs niveaux de listes de concaténations (une liste peut contenir une expression qui contient une liste), comme dans la figure 3.2.

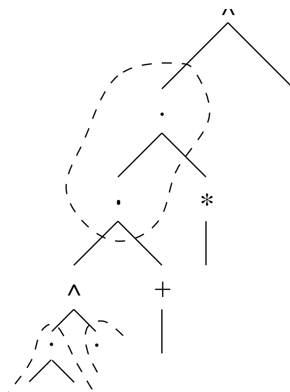

Fig. 3.2 – Groupement des concaténations en liste de concaténation

Lorsque l'on rencontre le premier élément d'une liste, on crée une structure de liste vide. Ensuite, les éléments seront ajoutés un par un à cette liste au fur et à mesure du parcours.

3.4.2 Mise en correspondance des parenthèses

Pendant ce parcours, on garde en mémoire un ensemble de piles de parenthèses, chaque pile correspondant à une couleur. La pile est gérée de la manière suivante :

- Lorsque l'on rencontre une parenthèse ouvrante, on l'empile sur la liste correspondant à sa couleur.
- Lorsque l'on rencontre une parenthèse fermante, on dépile une parenthèse, qui est donc la parenthèse correspondante. On ajoute sur chaque parenthèse un pointeur vers l'autre, ce qui permet à partir de ce moment de disposer des fonctions `Get_Opening` et `Get_Closing` sur les parenthèses.
- Lorsqu'on arrive à la fin d'une liste de concaténation, on vérifie que toutes les parenthèses sont bien fermées.

3.4.3 Restriction de la sémantique statique

La syntaxe des expressions régulières n'interdit pas des expressions comme $\langle a \rangle_1$, qui n'ont pourtant pas de sens, puisqu'on applique une contrainte de temps sur une expression (a) qui ne permet pas au temps de s'écouler. Ces contradictions peuvent pour la plupart être détectées statiquement, et leur élimination permettra une meilleure optimisation.

L'algorithme est assez simple : On sait facilement dire si une formule atomique permet au temps de s'écouler (faux pour a et ε, vrai pour $\underline{a}$). On peut étendre cette définition aux expressions quelconques de la manière suivante :

Définition 15 *L'ensemble $\mathcal{T}$ des expressions ne permettant pas l'écoulement du temps :*
- $\varepsilon \in \mathcal{T}$ *;*
- $a \in \mathcal{T}$*, pour tout symbole* a *;*
- *Si* $\varphi, \psi \in \mathcal{T}$*, alors,* $\varphi \vee \psi \in \mathcal{T}$
- *Si* $\varphi \in \mathcal{T}$ *et* ψ *quelconque, alors* $\varphi \wedge \psi \in \mathcal{T}$ *et* $\psi \wedge \varphi \in \mathcal{T}$ *;*
- *Si* $\varphi, \psi \in \mathcal{T}$*, alors* $\varphi \cdot \psi \in \mathcal{T}$ *et* $\psi \cdot \varphi \in \mathcal{T}$ *;*
- *Si* $\varphi \in \mathcal{T}$*, alors,* $\varphi^*, \varphi^+ \in \mathcal{T}$*.*

Notons que cette définition est donnée de manière à pouvoir décider facilement de l'appartenance d'une expression à $\mathcal{T}$. Elle ne correspond malheureusement pas à la définition intuitive : $\mathcal{T} = \{\varphi \mid \lambda([\![\varphi]\!]) \neq \{0\}$[1]$\}$. En revanche, on a le résultat suivant :

Théorème 5 *Pour toute expression régulière temporisée avec parenthèses colorées E,*

$$E \in \mathcal{T} \Rightarrow \lambda([\![E]\!]) = \{0\}$$

La vérification se fait dans le même parcours que la mise en correspondance des parenthèses. A chaque ouverture de parenthèses, on pose un marqueur sur la parenthèse ouvrante, disant par défaut que le temps ne peut pas s'écouler après celle-ci (valeur « faux »). On calcule à la volée et pour chaque noeud de l'arbre, son appartenance à $\mathcal{T}$. Lorsque l'on passe sur une expression de la même liste de concaténation permettant au temps de s'écouler, on change le marqueur de toutes les parenthèses ouvertes à « vrai ». Lorsque l'on rencontre une parenthèse fermante, on vérifie que le marqueur correspondant est bien à « vrai ».

3.4.4 Déplacement des parenthèses

Nécessité de la transformation

L'algorithme théorique donné dans [**?**] considère les cas ou les symboles sont précédés de temps, puisque les expressions sont définies à partir de a et $\underline{a}$. Néanmoins, avec un renommage judicieux, on peut définir un intervalle de temps quelconque : « $\theta : a \rightarrow \varepsilon; \underline{a}$ » a pour sémantique $\mathbb{R}^+$.

C'est la raison d'être de la macro `\time`, qui évite à l'utilisateur d'utiliser explicitement le renommage dans cette situation. Nous la noterons indifféremment `\time` ou τ par la suite.

Cela permet de redéfinir $\underline{a}$ comme `\time a`. Malheurseusement, l'algorithme appliqué tel quel à cette dernière expression est inefficace, puisqu'il compile une contrainte de temps pour le symbole `a`, comme dans la figure 3.3, alors qu'elle n'est pas nécessaire, et que l'on peut la compiler directement comme dans la figure 3.4.

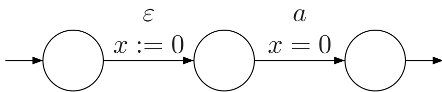

FIG. 3.3 – Automate non optimisé pour `\time a`

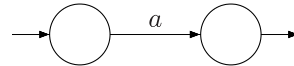

FIG. 3.4 – Automate optimisé pour `\time a`

Préliminaires

La solution est de déplacer les contraintes de temps lorsque c'est possible. On peut déplacer une parenthèse dans une liste de concaténation si on ne lui fait pas traverser une expression autorisant l'écoulement de temps. La correspondance des parenthèses est toujours maintenue puisque nous avons interdit les parenthèses autour d'expressions n'autorisant pas l'écoulement du temps.

[1]Un exemple d'expression vérifiant $\lambda([\![\varphi]\!]) \neq \{0\}$ qui n'est pas dans T est $(\underline{a} \vee x) \wedge (\underline{b} \vee x)$.

Théorème 6 *Si ω n'autorise pas l'écoulement du temps, et que les deux expressions suivantes sont valides, alors, elles sont équivalentes :*

$$\varphi \cdot \langle \omega \cdot \psi \quad \equiv \quad \varphi \cdot \omega \cdot \langle \psi$$

$$\varphi \cdot \omega \rangle_I \cdot \psi \quad \equiv \quad \varphi \rangle_I \cdot \omega \cdot \psi$$

(« $\langle$ » et « $\rangle_I$ » désignent des parenthèses ouvrantes et fermantes quelconques.)

Ce théorème va nous permettre de grouper les parenthèses, et donc de minimiser le nombre d'horloges utilisées.

Exemple

Prenons un exemple simple pour fixer les idées[2] : $\varphi = \underline{a} \cdot \lfloor b \cdot \langle c \cdot \underline{d} \cdot e \rfloor \cdot \tau \rangle \cdot f \cdot g$.

On peut le réécrire

$$\varphi = \tau \cdot a \cdot \lfloor b \cdot \langle c \cdot \tau \cdot d \cdot e \rfloor \cdot \tau \cdot f \rangle \cdot g$$

et déplacer les parenthèses pour obtenir

$$\varphi = \tau \cdot a \cdot \lfloor \langle b \cdot c \cdot \tau \cdot d \rfloor \cdot e \cdot \tau \cdot f \rangle \cdot g$$

La nouvelle expression présente plusieurs avantages :

- Les deux parenthèses ouvrantes ont été groupées, ce qui permettra, lors de la compilation, de ne créer qu'une seule horloge pour les deux contraintes,
- Chaque parenthèse fermante correspond à un symbole, et pourra être compilé par une contrainte de temps dans le garde d'une transition existante dans l'automate.
- La parenthèse fermante est rapprochée au maximum, ce qui permet d'obtenir un automate avec un nombre minimal d'horloges actives à un point donné.
- La contrainte exprimant que le temps ne peut pas s'écouler entre a et c peut être compilée avec la même horloge que celle des parenthèses.

Règle de transformation

Toutes les parenthèses se trouvant dans une suite d'expressions n'autorisant pas d'écoulement du temps doivent être déplacées vers la deuxième position la plus a gauche dans cette suite. (La plus à gauche qui ne soit pas précédé directement d'un τ)

Algorithme pratique

En pratique, cette optimisation peut se faire pendant la seconde passe, en même temps que la vérification de la correspondance des parenthèses.

Lors de cette passe, les opérateurs de concaténation binaires sont transformés en liste, et les expressions composant la liste sont ajoutées une par une en queue de liste.

Nous allons modifier la structure des expressions régulières en ajoutant la notion de suite d'événements n'autorisant pas l'écoulement du temps. Cette suite sera considérée comme un seul bloc dans la liste de concaténation, auquel seront attachées toutes les parenthèses se rapportant à l'un des élément de cet ensemble.

L'algorithme est alors assez simple : On garde en mémoire non plus une liste de concaténation, mais une liste plus un bloc en construction. Lorsque l'on rencontre une expression autorisant le passage du temps, on termine la construction du bloc, et on l'ajoute à la liste.

Par exemple, l'expression φ utilisée plus haut devient

$$\varphi = \tau \cdot \lfloor \langle (abc) \cdot \tau \cdot (de) \rfloor \cdot \tau \cdot (fg) \rangle$$

[2] Les intervalles ont été omis pour plus de clarté sur les parenthèses fermantes

3.5 Compilation en automate

La compilation en automates se fait de manière récursive. L'algorithme de transformation ressemble donc à

```
- Compiler les opérandes de l'opérateur de plus haut niveau.
- Appliquer cet opérateur aux automate(s) obtenu(s).
```

Le but sera ici d'effectuer un maximum d'optimisations, au prix d'un algorithme plus complexe que celui de [?].

3.5.1 Compilation des formules atomiques

Nous décrivons dans un premier temps la phase terminale de cette récursion : Les expressions ne contenant ni opérateurs ni parenthèses, c'est à dire, les symboles (`a`) ou ensembles de symboles (`[a b c]`).

Dans ce cas, l'automate est défini directement :

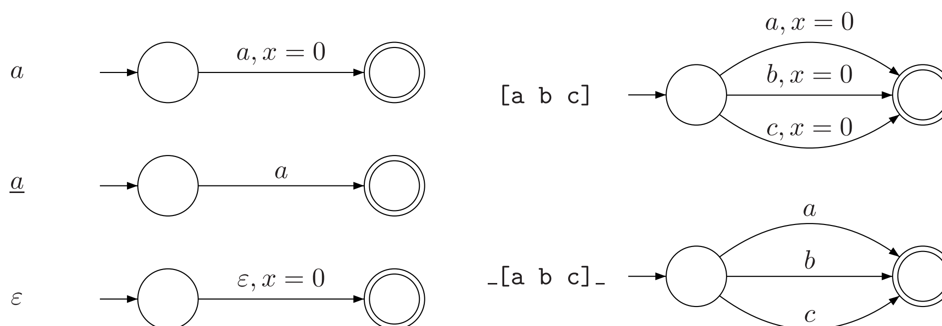

FIG. 3.5 – Automates correspondant aux expressions atomiques

La seule subtilité algorithmique consiste à déterminer quelle horloge tester dans les cas où on doit vérifier que le temps ne s'écoule pas. En effet, lorsqu'une parenthèse ouvrante se trouve sur le même symbole, une horloge est déjà initialisée, et il n'est pas nécessaire d'en allouer une nouvelle.

Ceci est en fait géré au niveau de la compilation d'atomes parenthésés.

3.5.2 Opérations de base

« Et » logique : produit cartésien

L'implémentation se fait en deux étapes : Un produit cartésien générique, implémenté au niveau de l'automate non temporisé, et un produit cartésien temporisé, qui est une encapsulation d'une instanciation de ce produit cartésien générique.

Le produit générique est paramètré par

– une fonction disant si deux transitions doivent être synchronisées. Dans notre cas, deux transitions sont synchronisées si les symboles sont égaux.
– Une fonction qui fusionne deux transitions. Ici, la fusion des transitions correspond au « et » logique des gardes.
– Une fonction qui fusionne deux états.

L'algorithme ne construit pas la totalité de l'automate produit, mais seulement les états atteignables. On commence par créer les états produits d'états initiaux, qui seront initiaux dans l'automate produit, et on les ajoute à la liste des états à traiter.

L'algorithme est alors une boucle : On choisi un état à traiter, on construit les transitions sortantes de cet état. On conserve un dictionnaire (implémenté par une table de hachage) associant les états produits déjà

construits aux paires d'états des automates de départ. On peut donc savoir si la destination de la nouvelle transition existe déjà, ou bien s'il faut la créer. Dans ce dernier cas, l'état créé est ajouté à son tour à la liste des états à traiter.

« Ou » logique : union d'automates

Le « ou » logique est une union d'automates. Pour ne garder qu'un état initial (le reste du code est prévu pour pouvoir fonctionner avec plusieurs états initiaux, mais ne l'utilise pas), on ajoute un état initial relié aux états initiaux des automates de départ par des ε-transitions.

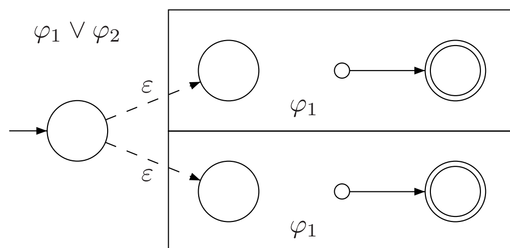

FIG. 3.6 – Union de deux automates.

Dans le cas où l'état de destination de cette ε-transition est final, il faut ajouter une contrainte de temps $x = 0$, pour ne pas ajouter l'intervalle de temps quelconque au langage. x est une horloge quelconque de l'un des deux automates Si il n'y en a pas, il faut en ajouter une.

Théorème 7 *Soit $A = (Q, C, \Delta, \Sigma, s, F)$ un automate temporisé quelconque reconnaissant le langage L.*

Si $s \notin F$, alors, l'automate $A' = (Q \cup \{i\}, C, \Delta \cup \delta, \Sigma, i, F)$, avec $\delta = (i, true, \emptyset, \varepsilon, s)$ reconnaît le langage L.

Démonstration :
Il est clair que toute exécution acceptée par A l'est aussi par A', car il suffit d'ajouter $(i, v_0) \xrightarrow{\varepsilon} (s, v_0) \xrightarrow{\cdots} \ldots$ en tête.

Réciproquement, si une exécution est acceptée par A', alors, elle commence nécessairement par un éventuel écoulement de temps suivi du passage de l'ε-transition : $(i, v_0) \xrightarrow{t} (i, v_1) \xrightarrow{\varepsilon} (s, v_1) \xrightarrow{\cdots} \ldots$. Alors, l'écoulement de temps dans l'état initial i peut être fait de la même façon dans s : $(s, v_0) \xrightarrow{t} (s, v_1) \xrightarrow{\cdots} \ldots$
□

L'opérateur +

Pour chaque transition vers un état final, on ajoute une copie de cette transition vers l'état initial, en réinitialisant toutes les horloges de l'automate.

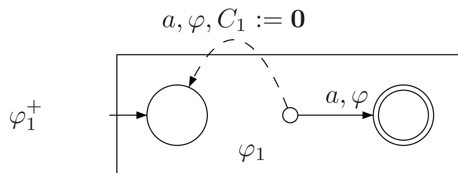

FIG. 3.7 – Traduction de l'automate de φ^+

L'opérateur $*$

Dans [?], l'expression φ^* est réécrite en $\varepsilon \vee \varphi^+$, et traduite à partir des algorithmes de transformation de $\varphi \vee \psi$ et de φ^+. Cette solution est simple, mais peu optimisée.

Dans la plupart des cas, il est suffit simplement de marquer les états initiaux de l'automate de φ^+ comme finaux. Ainsi, l'automate reconnaîtra également la chaîne vide. Le problème est qu'il est possible que la trace d'exécution de l'automate repasse par l'état initial, et le rendre final peut ajouter d'autres mots que la chaîne vide au langage. Il convient donc de formaliser une condition suffisante pour pouvoir légitimement effectuer cette optimisation.

Théorème 8 *Soit $A = (Q, C, \Delta, \Sigma, s, F)$ un automate temporisé reconnaissant le langage L.*
Si $\forall (q, \varphi, \rho, a, q') \in \Delta, q' \neq s$, alors, l'automate $A' = (Q, C, \Delta, \Sigma, s, F \cup \{s\})$ reconnaît le langage $L \cup \varepsilon$.

Démonstration :
On procède par double inclusion. Il est trivial de vérifier que ε est reconnu par l'automate, et tout mot de L étant reconnu par A, la trace d'exécution de ce mot est également acceptée par A'

Réciproquement, soit m un mot accepté par A'. Si $m = \varepsilon$, alors, $m \in L \cup \varepsilon$. Dans le cas contraire, la trace d'exécution de m est $(q_0, v_0) \xrightarrow{z_1} (q_1, v_1) \xrightarrow{z_2} \cdots \xrightarrow{z_n} (q_n, v_n)$, avec $q_n \in F \cup \{s\}$. La dernière transition est $(q_{n-1}, \varphi, \rho, a, q_n)$, Or, par hypothèse, $q_n \neq s$, donc, $q_n \in F$, donc, l'exécution de m est aussi acceptée par A, ce qui conclue la preuve. □

Ce théorème permet de compiler $\varepsilon \vee \varphi^+$ en rendant l'état initial final.

Dans le cas contraire, il faut ajouter un nouvel état initial pour remplacer l'ancien, et une ε-transition :

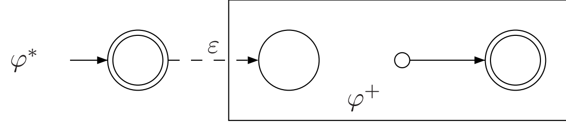

Fig. 3.8 – Compilation de φ^* dans le cas général

On justifie cette construction comme suit :

Théorème 9 *Soit $A = (Q, C, \Delta, \Sigma, s, F)$ un automate temporisé quelconque reconnaissant le langage L.*
L'automate $A' = (Q \cup \{i\}, C, \Delta \cup \delta, \Sigma, x, F \cup \{x\})$, avec $\delta = (i, x = 0, \emptyset, \varepsilon, s)$ et x est une horloge quelconque de C (Si $C = \emptyset$, il faut en ajouter). reconnaît le langage $L \cup \varepsilon$.

Démonstration :
ε est reconnu par l'automate car l'état initial est final. Si m est un mot de L, alors, m a pour trace d'exécution, acceptée par A, $(s, v_0) \xrightarrow{z_1} (q_1, v_1) \xrightarrow{z_2} \cdots \xrightarrow{z_n} (q_n, v_n)$. En ajoutant une ε-transition au début, on obtient $(x, v_0) \xrightarrow{\varepsilon} (s, v_0) \xrightarrow{z_1} (q_1, v_1) \xrightarrow{z_2} \cdots \xrightarrow{z_n} (q_n, v_n)$

Réciproquement, soit m un mot accepté par A'. Si $m = \varepsilon$, alors, $m \in L \cup \varepsilon$. Dans le cas contraire, la trace d'exécution de m est $(x, v_0) \xrightarrow{z_1} (q_1, v_1) \xrightarrow{z_2} \cdots \xrightarrow{z_n} (q_n, v_n)$. La première transition est forcément l'ε-transition, qui ne peut être franchie qu'au temps 0, donc, l'exécution est en fait $(x, v_0) \xrightarrow{\varepsilon} (s, v_0) \xrightarrow{z_2} \cdots \xrightarrow{z_n} (q_n, v_n)$ qui est également acceptée par A. □

L'algorithme est donc le suivant :
- Calculer l'automate A de φ,
- `x` = « il existe des transitions vers l'état initial de A »,
- Appliquer l'algorithme de transformation de φ^+ sur A.
- Si `x` = « vrai », appliquer la transformation générale de la figure 3.8.
- Si `x` = « faux », appliquer l'optimisation du théorème 8.

3.5.3 Concaténations et parenthèses

Les notions de parenthèse colorées et de concaténations sont intimement liées, puisque nous avons choisi de n'autoriser les parenthèses qu'à l'intérieur d'une liste de concaténation. On peut cependant utiliser des listes de concaténation de longueur 1.

Atomes parenthésés

Un atome parenthésé (définition 13) est constitué d'un ensemble de parenthèses ouvrantes, d'une expression, et d'un ensemble de parenthèses fermantes, qui seront traitées successivement.

Parenthèses ouvrantes Si la liste de parenthèses ouvrantes est non vide, on alloue une nouvelle horloge, et on l'associe à chaque paire de parenthèses colorées de la liste. On garde également un compteur donnant le nombre de parenthèses ouvrantes non fermées pour cette horloge. Cette information est stockée dans l'objet « paire de parenthèses », avec une sémantique de partage pour que les modifications faites sur le compteur soient visibles par toute la liste.

Une horloge est également allouée si l'expression contenue dans l'atome n'autorise pas l'écoulement du temps.

Expression L'expression est traduite comme une expression ordinaire, sauf dans le cas particulier des expressions n'autorisant pas le passage du temps qui subiront un traitement particulier précisé en section 3.5.3.

Parenthèses fermantes Les parenthèses fermantes correspondent à des gardes à ajouter sur des transitions.

Le problème est qu'il ne faut ajouter ces gardes que sur la dernière transition franchie par une exécution. Dans le cas où il n'y a aucune transition quittant l'état final de l'automate, on peut simplement ajouter la garde sur chaque transition vers l'état final (figure 3.10). Dans le cas contraire, il faut ajouter un nouvel état final (figure 3.9).

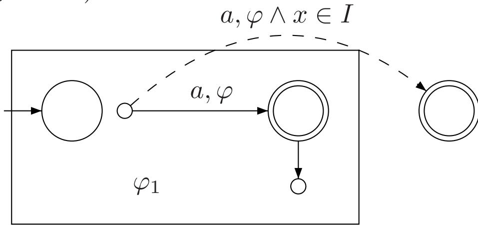

Fig. 3.9 – L'état final doit être ajouté

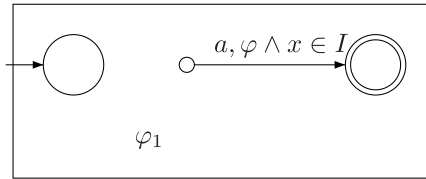

Fig. 3.10 – L'état final ne doit pas être ajouté

Il faut également traiter le cas particulier des états à la fois initiaux et finaux. En effet, ces automates reconnaissent la chaîne vide, dont l'exécution ne franchit aucune transition, donc aucune garde. Si l'on écrit $\langle\varphi\rangle_1$ par exemple, même si $\varepsilon \in [\![\varphi]\!]$, on ne peut pas avoir $\varepsilon \in [\![\langle\varphi\rangle_1]\!]$ car $\lambda(\varepsilon) = 0 \neq 1$.

Dans le cas où l'état initial est en même temps final, on applique au préalable la transformation de la figure 3.11.

Reste à formaliser ce résultat.

Théorème 10 *Soit $A = (Q, C, \Delta, \Sigma, s, F)$ un automate temporisé reconnaissant le langage L, x une horloge de A ne subissant aucune initialisation dans A (s'il n'y en a pas, nous en ajoutons une.) et I un intervalle borné par des entiers naturels.*

Si $\forall (q, \varphi, \rho, a, q') \in \Delta, q \notin F$, et si $s \notin F$, alors, l'automate $A' = (Q, C, \Delta', \Sigma, s, F)$ avec

$$\Delta' = \{(q, \varphi, \rho, a, q') \in \Delta \mid q' \notin F\} \cup \{(q, \varphi \wedge x \in I, \rho, a, q') \mid q' \in F \text{ et } (q, \varphi, \rho, a, q') \in \Delta\}$$

reconnaît le langage $L \cap \{x \mid \lambda(x) \in I\}$.

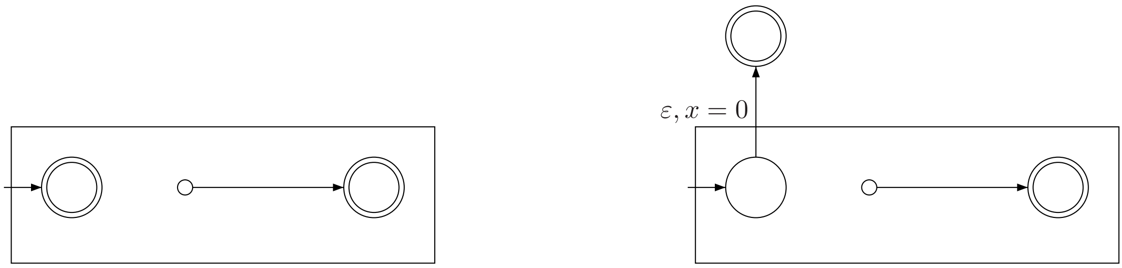

Fig. 3.11 – Transformation rendant impossible une exécution sans transition

Démonstration :
Ici encore, on procède par double inclusion.

Soit m un mot de $L \cap \{x \mid \lambda(x) \in I\}$. Comme $m \in L$, m est reconnu par A, et l'exécution se termine par une transition vers un état final. Comme x n'est jamais réinitialisé, l'interprétation de x au passage de la dernière transition T est $\lambda(m)$. On sait par ailleurs que $\lambda(m) \in I$, donc, la garde de la transition de A' correspondant à T est vérifié.

Hormis la dernière transition, il n'est pas possible que l'exécution de m contienne une transition $(q, \varphi, \rho, a, q')$ vers un état final, car alors, la transition suivante serait de la forme $(q', \varphi, \rho, a, q'')$ avec $q' \in F$, ce qui est contraire aux hypothèses. Donc, chaque transition de l'exécution de m sauf la dernière existe également dans l'automate A'.

Nous pouvons donc conclure que toutes les étapes de l'exécution de m sont possibles dans A'.

Réciproquement, soit m un mot reconnu par A'. Son exécution se termine forcément par une transition vers un état final (puisque l'état initial n'est pas final), donc, par une transition de la forme $\{(q, \varphi \wedge x \in I, \rho, a, q') \mid q' \in F\}$. Il est donc clair que $\lambda(m) \in I$. Les transitions de A' étant les mêmes que celles de A avec des gardes plus restrictives, une exécution acceptée par A' le sera également par A, et donc, finalement, $m \in L \cap \{x \mid \lambda(x) \in I\}$. □

Le cas général avec ajout d'état final est donné dans [**?**] et n'est par rappelé formellement ici.

Concaténation standard

La concaténation standard est une suite d'atomes parenthésés. On les construit un par un, puis on relie les états finaux aux états initiaux du suivant. On se ramène donc au cas de la concaténation binaire.

L'algorithme donné dans [**?**] ajoute une transition vers l'état initial du second automate pour chaque transition vers l'état final du premier :

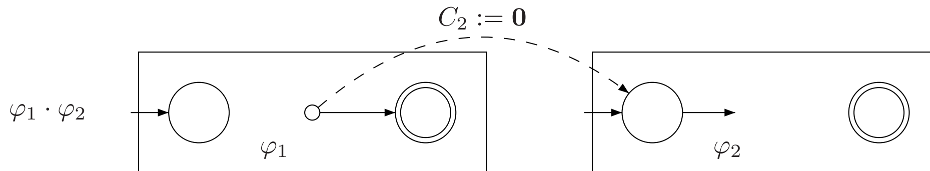

Fig. 3.12 – Algorithme simple de concaténation de deux automates

Cet algorithme ne peut malheureusement pas s'appliquer au cas ou un état est à la fois initial et final (même dans le cas non temporisé). La figure 3.13 en donne un contre-exemple : Il est clair que le mot c devrait être reconnu par le langage, mais que le plus court mot reconnu est $a \cdot b \cdot c$.

Ceci ne remet pas en cause la correction de [**?**], puisque les autres formules de transformations n'engendrent jamais de tels cas, mais l'optimisation que nous avons réalisée sur la transformation de l'étoile nous oblige à traiter ce cas.

Dans le cas d'un état initial et final, on va rajouter une transition de l'état en question vers chaque état suivant l'état initial du second automate :

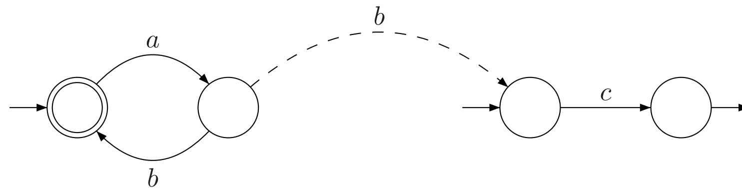

FIG. 3.13 – Contre-exemple pour l'algorithme simple de concaténation

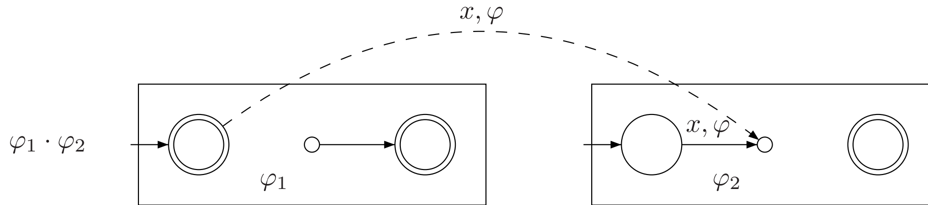

FIG. 3.14 – Cas particulier d'un état initial et final

Malheureusement, cet algorithme n'est pas correct dans le cas où l'état initial du second automate est également final, comme le montre la figure 3.15.

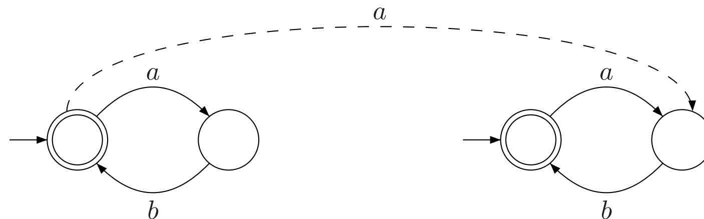

FIG. 3.15 – Contre-exemple pour l'algorithme simple de concatenation

En effet, l'automate généré ne reconnaît pas la chaîne vide, alors qu'il le devrait.

Dans ce cas, on applique la même transformation que dans la figure 3.11.

Liste de concaténation sans passage de temps

Dans le cas d'une liste de concaténation sans passage de temps, on construit les automates des éléments de la liste, et on les concatène comme pour une concaténation standard. Ensuite, on examine chaque transition une par une, et on remplace chaque gardes par $x = 0$, où x est une horloge initialisée au début du bloc. Si une parenthèse ouvrante se trouve juste avant le bloc, on peut réutiliser l'horloge correspondante, et sinon, il faut en allouer une nouvelle.

3.6 Aspect pratique

Le langage choisi pour la partie algorithmique est l'Ada 95, pour la simple raison que c'est mon langage préféré (Typage fort, bonne gestion des exception, lisibilité, ...). Le compilateur utilisé est GNAT 3.14P (Les versions antérieures comportent un bug et ne compilent pas le projet correctement).

Le programme de transformation proprement dit est encapsulé dans un script qui fait toutes les transformations et appelle KRONOS. Celui-ci est écrit en Perl. D'autres scripts, dont les scripts de test automatique sont écrits en shell sh.

Les « booch components » (équivalent Ada de la STL de C++) sont utilisés pour les différentes structures d'ensembles, listes, et tables de hachage.

Les parties analyse lexicale et syntaxique sont faites avec Aflex et Ayacc, équivalents Ada de Flex et Yacc.

Pour la sortie au format BCG, une interface Ada a été écrite pour la bibliothèque Bcg_user, écrite en C (qui ne compte que quelques fonctions).

L'ensemble du projet est placé sous CVS pour garder un historique du développement, et faciliter le travail sur plusieurs machines.

La compilation du projet est géré par un ensemble de Makefile.

Le projet est écrit sous GNU/Linux (sur plateforme i686). Il devrait être portable vers d'autres systèmes UNIX (non complètement testé), mais aucun effort n'a été fait par rapport à la portabilité en dehors du monde UNIX.

Le tableau suivant donne un aperçu de la taille des modules du projet, en nombre de lignes de code (comptées brutalement avec wc, donc, incluant les commentaires et les lignes vides).

Module	Lignes de code
Portions communes	1 408
Structure d'expression	2 132
Parseur d'expressions	2 518
Structure d'automate	4 568
Parseur d'automates	1 023
Compilation d'expression en automate	1 300
Encapsulation de la librairie BCG en Ada	110
Ajouts aux « booch components »	91
Petits programmes de tests	459
Scripts divers	457
Total	14 066

Tab. 3.1 – Nombre de lignes de codes des différents modules du compilateur.

Chapitre 4

Étude de cas

4.1 Introduction

4.2 Intégration du compilateur d'expressions dans Kronos

4.2.1 Aspect théorique

Problème du langage vide

KRONOS est un outil capable de vérifier une propriété exprimée en logique TCTL sur un automate. Il peut vérifier que cette propriété est vraie pour tous les états, mais également décider de l'atteignabilité d'un état vérifiant cette propriété à partir d'un état vérifiant une propriété initiale. (Pour une documentation technique, se référer à [?], et pour l'aspect théorique, voir [?]).

En prenant comme propriétés initiales et finales des prédicats simples, (par exemple, `INIT` et `FINAL`), on décide donc de l'existence d'une trace d'exécution partant d'un état marqué par le prédicat `INIT`, et terminant sur un état marqué par `FINAL`. C'est exactement l'existence d'un comportement accepté par l'automate, où encore, le problème du langage vide.

Intersection de langages

L'intersection de langages correspond au produit cartésien synchronisé, ce que KRONOS fait parfaitement. Il faudra simplement donner une synchronisation correcte pour faire le produit.

Vérification d'une propriété

Malheureusement, le problème de l'inclusion des langages réguliers temporisés est indécidable, et leur complémentation n'est pas calculable. (voir par exemple [?])

Pour palier à ce problème, nous allons exprimer les propriété inacceptables de notre système, et vérifier que ces propriétés sont irréalisables.

On note φ le langage acceptable pour le système, et ψ le langage effectivement implémenté. La propriété que nous allons exprimer correspondra donc à $\overline{\varphi}$. On appelle alors KRONOS pour le problème du langage vide sur $\psi \cap \overline{\varphi}$. La figure 4.1 illustre cet démarche. Si la propriété est vérifiée, l'intersection sera vide, et KRONOS répondra que l'atteignabilité a échoué. Dans le cas contraire, on obtient une trace prouvant l'atteignabilité, qui est un contre-exemple pour la propriété.

4.2.2 Aspect pratique

Il nous reste à mettre en pratique ces théories. Nous ne reviendrons pas sur la compilation de l'expression en automate, qui a déjà été étudiée en détails. Une vue d'ensemble de cette mise en pratique est illustrée par la figure 4.2.

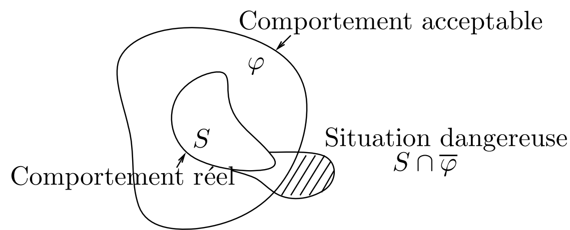

Fig. 4.1 – Mauvaises propriétés

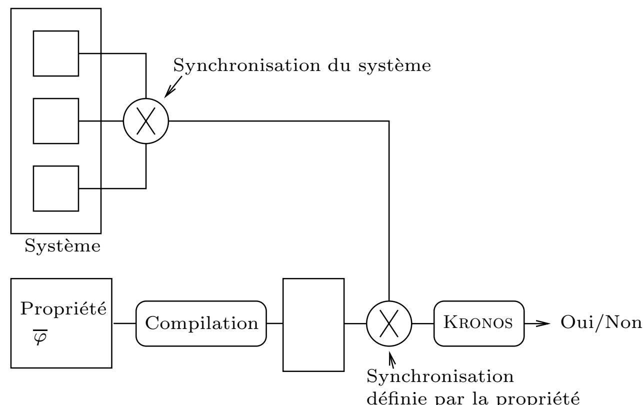

Fig. 4.2 – Vue d'ensemble de la vérification avec Kronos

Problème de la synchronisation

Kronos gère la synchronisation du produit cartésien d'une manière assez simple : Chaque automate contient une ligne commençant par `#sync`, et donnant la liste des symboles pour lesquels il doit y avoir synchronisation.

Il y a deux manières de faire le produit cartésien : On peut soit générer l'automate produit dans un fichier, puis utiliser ce fichier pour la vérification, soit donner directement au model-checker tous les fichiers, et le produit sera alors fait « à la volée », pour éviter une éventuelle explosion d'états.

Ce qui nous pose problème est que la synchronisation est donc une donnée propre à l'automate, et non à l'opérateur de produit cartésien. Or, notre algorithme nécessite deux produits différents, avec synchronisations différentes. Il n'existe pas, à l'heure actuelle, de solution permettant de faire ces deux produits à la volée.

Génération du système global : On laisse donc l'utilisateur placer les lignes `#sync` dans les composants du système pour décrire correctement le système, et sans lien avec la ou les propriétés à vérifier. La génération du système global à partir de ses composants doit être faite statiquement, et le résultat est stocké dans un fichier.

Resynchronisation pour la vérification. Il faut alors modifier la synchronisation pour qu'elle corresponde à l'alphabet de la propriété. En effet, si l'on écrit une propriété comme

```
#alphabet a b c
#expression
  a . c
```

on exprime en fait « Il y a une occurrence de a suivie directement de c, sans qu'il n'y ai eu de b entre les deux ». Si le système n'est pas synchronisé sur b, alors, il peut émettre un b comme événement interne, ce qui fausse totalement le résultat.

C'est la raison d'être du script `synchronize.pl`, qui prends deux fichiers en arguments, et positionne le champ `#sync` de l'un sur la valeur de l'autre. (Tout le travail est fait par substitution d'expressions régulières, pour enlever les commentaires et trouver l'emplacement du champ).

La vérification proprement dite. Nous disposons donc de deux automates : L'automate du système, et celui de la propriété. Il suffit d'appeler Kronos dans le mode atteignabilité pour obtenir la réponse. Lors de la compilation d'expression, nous avons marqué les états initiaux par `INIT` et finaux par `FINAL`. L'automate du système n'a pas d'état final, puisque son exécution est potentiellement infinie. En revanche, on peut vouloir spécifier un état initial. L'état initial global sera le produit des états marqués `INIT` (pour la propriété) et des états initiaux du système, c'est à dire, ceux vérifiant la propriété « INIT and (<état initial du système>) ».

Résumé Le script `trev.pl` fait tout ceci pour l'utilisateur :
- Compilation de l'expression en automate,
- Génération de l'automate global du système,
- Re-synchronisation de l'automate du système,
- Calcul d'atteignabilité.

4.3 Un exemple simple : Le passage à niveau

4.3.1 Description du système

Le problème du passage à niveau est un exemple assez classique de vérification de système temporisé. L'exemple est tiré de [?]. Un train peut passer entre les barrières. Un contrôleur détecte son arrivée et commande les barrières.

On fait certaines hypothèses sur le comportement du train et celui des barrières, et on connaît le temps de réponse du contrôleur. La propriété essentielle est qu'il n'est pas possible que le train passe sans que les barrières ne soient baissées.

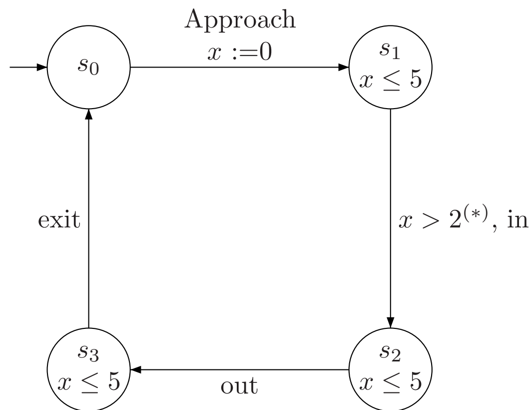

Fig. 4.3 – Train

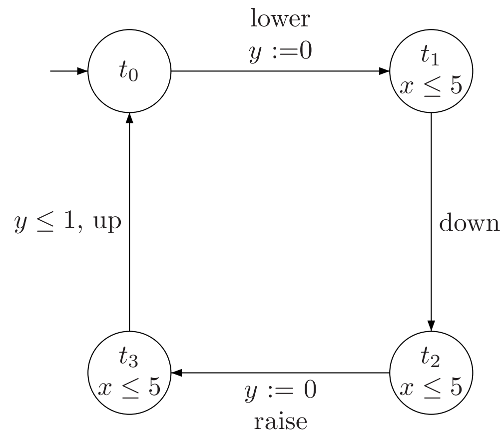

Fig. 4.4 – Barrières

4.3.2 Propriétés à vérifier

Le train ne passe pas entre les barrières ouvertes

Comme annoncé plus haut, la propriété la plus importante est le fait que le train ne puisse pas traverser les barrières quand celles-ci sont ouvertes. En d'autre terme, il ne doit pas y avoir successivement d'occurrence de `approach` puis de `in` sans qu'il n'y ai eu d'occurrence de `down` entre les deux.

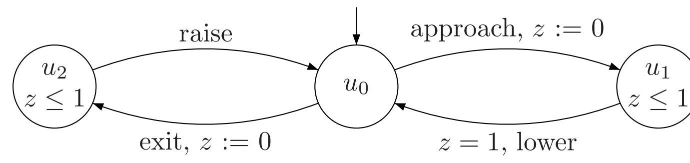

FIG. 4.5 – Contrôleur

La mauvaise propriété est donc :

```
#alphabet approach in down
#expression
  (_[...]_)* . _approach_ . _in_
```

Il est intéressant de remarquer que cette propriété n'est vérifiée que grace à l'aspect temporisé de l'automate, mais ne fait pas intervenir le temps directement.

Sur cet exemple, la propriété est réellement très simple à exprimer, alors qu'elle serait beaucoup moins évidente avec une logique temporelle.

Notons tout de même un piège : le choix du champ `#alphabet` n'est pas si anodin qu'il n'en a l'air. En effet, on serait tenté d'y mettre tous les symboles utilisés, de la même manière que l'on déclare les variables dans un langage classique. Or, omettre l'événement `down` signifierait le considérer comme un événement interne du système, et la mauvaise propriété exprimerait alors : « Il y a une occurrence de `approach` suivie d'une occurence de `in` », ce qui est toujours possible.

Il s'écoule suffisamment de temps entre l'abaissement de la barrière et le passage du train

On peut ne pas se satisfaire de la propriété de sûreté précédente et exiger que non seulement la barrière soit baissée lors du passage du train, mais qu'elle le soit depuis suffisamment longtemps.

La mauvaise propriété est donc que l'événement `down` soit suivi de l'événement `in` dans un intervalle de temps de moins de x secondes (pour fixer les idées, prenons $x = 1$). Elle s'écrit comme suit :

```
#alphabet down in
#expression
(_[...]_)* . _down_ . < _in_ >[0, 1)
```

Elle n'est malheureusement pas vérifiée : Les seules hypothèses que nous faisons sur l'environnement sont que le contrôleur met 1 seconde à réagir, la barrière met au plus 1 seconde à se baisser, alors que le train met plus de 2 secondes à arriver. Il est donc possible que le train arrive juste avant que la barrière ne se ferme.

Si on modifie la « vitesse » du train, en changeant la borne inférieure du délai d'arrivée du train (marqué $^{(*)}$ sur la figure 4.3), on peut rendre la propriété vraie, comme le montre le tableau 4.1.

Propriété \ Temps d'arrivée	$x > 2$	$x \geq 3$	$x > 3$
`_down_ . < _in_ >[0, 1)`	Faux	Vrai	Vrai
`_down_ . < _in_ >[0, 1]`	Faux	Faux	Vrai
`_down_ . < _in_ >[0, 2)`	Faux	Faux	Faux

TAB. 4.1 – Résultats du model-checker en fonction du temps d'arrivée du train.

Ces résultats montrent également la pertinence des égalités strictes et non strictes, qui ont une réelle influence sur le résultat du model-checker[1].

[1] C'est en réalité un problème dans la modélisation des systèmes temporisés, car bien que ces différences soient une réalité mathématique, elles ne correspondent pas à une réalité physique.

4.4 Force et faiblesses des expressions régulières temporisées.

Bien qu'étant théoriquement équivalentes aux automates, les expressions régulières sont différentes dans leur utilisation.

La situation est semblable à celle de la théorie de la calculabilité : Tous les langages de programmations sont en théorie équivalents, mais certains sont tout de même plus agréables que d'autres, et chaque langage est plus ou moins adapté à chaque situation.

4.4.1 Faiblesses

Impossibilité de nommer un état.

Les expressions régulières décrivent le comportement extérieur d'un système. La notion d'état y est donc totalement absente, et il n'est pas possible d'y faire référence dans une propriété. Or, il est assez fréquent que l'on souhaite vérifier des propriétés du type « Si le composant A est dans l'état x, alors, le composant B est dans l'état y », comme par exemple, dans l'exemple du passage à niveau, « Si le train est entre les barrières, alors, les barrières sont fermées ».

Logique simple, mais faible

Un autre problème est lié au fait que les expressions régulières travaillent en temps linéaire. On ne fait pas la différence entre $a \cdot (b \vee c)$ et $(a \cdot b) \vee (a \cdot c)$. Par contre, en utilisant une logique « branching time » sur les automates, ces comportements sont différents :

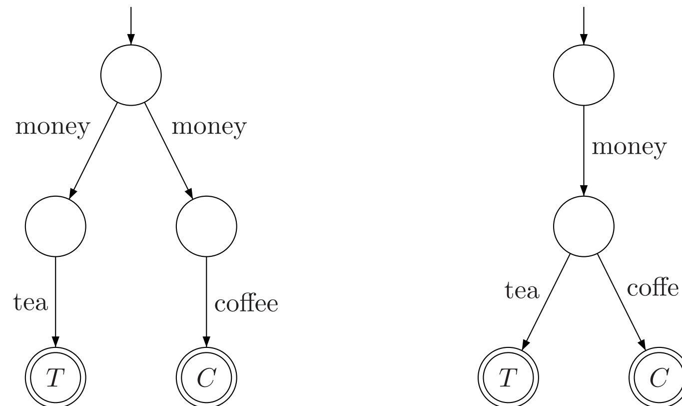

Fig. 4.6 – Deux automates reconnaissant le même langage

Dans le premier cas, l'utilisateur introduit une pièce, et à ce moment, la machine fait un choix interne et décide si elle servira du thé ou du café, et dans le second, le choix est fait au dernier moment. (c'est le comportement correct attendu d'un distributeur de boisson ...)

En TCTL, on peut distinguer les deux automates avec une propriété comme $\text{init} \Rightarrow \forall\diamond T$, qui est vrai sur le second, mais pas sur le premier.

Avec des expressions régulières, les deux systèmes sont indiscernables, et dans ce cas particulier, on ne pourra pas vérifier que l'utilisateur aura effectivement le choix de la boisson.

Impossibilité d'exprimer « enable »

Un autre problème du même ordre est l'impossibilité d'exprimer le fait qu'une transition est autorisée : On observe le comportement extérieur, on sait si un événement arrive ou bien s'il n'arrive pas, mais on ne peut pas savoir si « il aurait pu arriver ». Dans le cas d'un serveur, on veut en général exprimer la disponibilité de ce serveur. Une propriété de vivacité typique est « Le serveur redevient disponible au plus 5

secondes après avoir reçu une requête », ce qui revient à dire que le serveur se trouve dans un état d'où part une transition portant l'étiquette « recevoir une requête », et que la garde de cette transition est vérifié.

4.4.2 Atouts

Légèreté de la syntaxe

Les expressions régulières ont été conçues pour être écrites dans un format textuel. Les automates sont plus adaptés à une syntaxe graphique, et il est clair que leur syntaxe textuelle est beaucoup plus lourd et moins lisible. Ce défaut des expression est cependant comblé si on utilise un éditeur graphique.

Clareté de la sémantique

Sauf dans le cas de propriété très spécifique (utilisation du renommage et surtout de l'intersection), il est en général relativement aisé de donner une sémantique intuitive à une expression régulière. En outre, il est rare d'exprimer le langage vide sans s'en rendre compte. (Ce qui poserait problème dans l'utilisation des expressions régulières comme mauvaises propriétés, car alors, la propriété serait toujours vérifiée.)

4.4.3 Conclusion

Les expressions régulières temporisées peuvent donc être considérées comme un langage de haut niveau. Elles apportent un certain confort d'utilisation par rapport aux automates, mais ne permettent pas une description interne aussi fine que ces derniers.

En reprenant les études de cas réalisées avec Kronos(par exemple, [**?**], ou [**?**]), on se rends compte que la véritable difficulté des problèmes de vérification est en général la description du système proprement dite. Les propriétés à exprimer sont en général soit très simples soit trop pointues pour être exprimées avec des expressions régulières.

Conclusion et perspectives

Ce travail consistait en une étude de concepts théoriques non triviaux comme les automates temporisés et les expressions régulières temporisées, et en un transfert de résultats théoriques vers un prototype d'outil dans l'optique d'être intégré dans un système pratique. Comme bien souvent, l'influence de la théorie vers la pratique n'a pas été unidirectionnelle, et les problèmes découverts dans l'implémentation ont déclenché de nouvelles considérations théoriques.

Le point de départ du travail était l'idée de parenthèses colorées en remplacement de l'intersection. Les premières réflexions sur l'analyse de ces expressions ont mené à des décisions sur la structure de la syntaxe. La plus grande partie du travail a été la conception et la réalisation du compilateur d'expressions vers les automates, avec des versions pratiques des constructions théoriques. Cette expérience a été très enrichissante, et la vue de l'ensemble de la chaîne menant des expressions régulières à la vérification avec Kronos a été un certain aboutissement.

Pour la suite, une étude plus systématique des classes de propriétés utilisées pour la vérification des systèmes hybrides est requise. En refaisant les différentes études de cas, il semble que les propriétés fassent intervenir les états au même titre que les transitions. Cela nécessite probablement des expressions sur le monoïde mixte signal-événement. D'autres directions de recherches seraient une optimisation plus poussée de la transformation pour générer un espace d'états aussi petit que possible afin de faciliter la vérification, et l'étude de la « convivialité » du langage, en introduisant diverses sortes de sucres syntaxiques.

Il serait également intéressant de continuer les recherches pour introduire la notion d'expressions régulières temporisées dans des domaines autres que la vérification, comme par exemple les bases de données temporisées.

Annexe A

Sources Kronos des composants du passage à niveau

A.1 Automate du train

```
// Train
#states 4
#trans  4
#clocks 1 X
#sync approach exit

state: 0
invar: TRUE
trans:
TRUE => approach; RESET{ X }; goto 1

state: 1
invar: x <= 5
trans:
TRUE => in ; ; goto 2
// x > 2 => in ; ; goto 2

state: 2
invar: x <= 5
trans:
TRUE => out; ; goto 3

state: 3
invar: x <= 5
trans:
TRUE => exit; ; goto 0
```

A.2 Automate de la barrière

```
// Gate
#states 4
#trans  4
#clocks 1 y
```

```
#sync lower raise

state: 0
invar: TRUE
trans:
TRUE => lower; RESET{ y }; goto 1

state: 1
invar: y <= 1
trans:
TRUE => down ; ; goto 2

state: 2
invar: TRUE
trans:
TRUE => raise; RESET{ y }; goto 3

state: 3
invar: y <= 5
trans:
y > 1 => up; ; goto 0
```

A.3 Automate du contrôleur

```
// Controler
#states 3
#trans  4
#clocks 1 z
#sync approach lower raise exit

state: 0
invar: TRUE
trans:
TRUE => approach; RESET{ z }; goto 1
TRUE => exit; RESET{ z }; goto 2

state: 1
invar: z <= 1
trans:
z = 1 => lower ; ; goto 0

state: 2
invar: z <= 1
trans:
TRUE => raise; ; goto 0
```

Annexe B

Implementation Manual

B.1 Introduction

B.1.1 Overview of the software

The software is an implementation of the Kleene theorem for timed automata. Its goal is to transform expressions to automata, and automata to expressions.

B.1.2 Architecture of the software

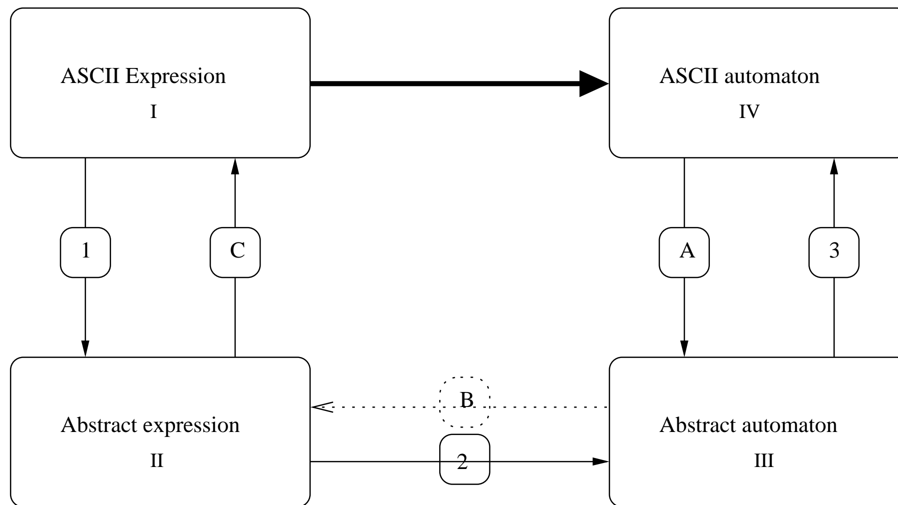

Figure B.1: Architecture of the compiler

The transformation B is not yet implemented.

The goal is to convert an ASCII file containing a regular expression to an automaton in KRONOS' format, and the opposite transformation.

To concentrate on algorithm and write a maximum amount of format independent code, I chose to make 3 passes : Parsing, transforming, and ASCII generation. The piece of software will thus need 4 data structures and 6 transformations, as shown in figure B.1.

ASCII Regular expressions (I) are defined in section B.3. They don't have a real data structure, but are only ASCII files.

Parsing expressions (1) use AFLEX and AYACC for lexical and syntactical context insensitive analysis. A second pass is needed for context sensitive analysis, to match the color brackets.

Abstract regular expressions (II) are represented by a tree in memory. A library is written and gives the primitives and the structure for building trees. The grammar described in section B.2 is checked when the tree is built and transformed.

Transformations (2 and B) in both ways are done from one abstract structure to the other. No ASCII is generated at this point.

Parsing automaton (A) is mainly a reuse of KRONOS' work. It also use AFLEX and AYACC.

ASCII Automata (IV) used by this program is only a subset of the automata used by KRONOS. They only use clock constraints on transitions, and the only clock manipulation allowed is reseting.

ASCII generation (C and 3) are the easiest parts. They use recursive algorithms.

B.2 ASCII Regular Expressions

This document describes the ASCII notation for timed regular expressions, in the form of a attribute grammar.

The following typographical conventions are used:

- *Non terminal symbol* (Grammar internals)
- `Terminal symbols` (What you will effectively write)
- ↑*Synthesized attributes*
- ↓*Inherited attributes*

B.2.1 Basic symbols

The basic notions of timed regular expressions are symbols, and colored brackets. We also need to define intervals to define the brackets.

We use an implicit renaming: For any symbol `a`, we define a set {`a`, `a'`, `a''`, ...}, all mapped to the symbol `a` by renaming.

INTERVAL ⟶ *LOWER_BOUND* ↑*lbound* `,` *UPPER_BOUND* ↑*ubound*
condition : $ubound \geq lbound$

INTERVAL ⟶ `NUMBER`

Notation: l for $[l, l]$

LOWER_BOUND ↑(**open**, *value*) ⟶ `(NUMBER` ↑*value*

LOWER_BOUND ↑(**closed**, *value*) ⟶ `[NUMBER` ↑*value*

UPPER_BOUND ↑(**open**, *value*) ⟶ `NUMBER` ↑*value* `)`

UPPER_BOUND ↑(**closed**, *value*) ⟶ `NUMBER` ↑*value* `]`

QUOTED_IDF ⟶ *IDF LIST_QUOTES*

SYMBOLS ⟶ _ *QUOTED_IDF* _

Any interval of time followed by the symbol

SYMBOLS ⟶ *QUOTED_IDF*

Stands for <_CONST_>0

SYMBOLS ⟶ _ *LIST_SYMBOLS* _

Any interval of time followed by any of the symbol

SYMBOLS ⟶ *LIST_SYMBOLS*

Stands for <_ LIST_SYMBOLS _>0

SYMBOLS $\longrightarrow$ \epsilon *LIST_QUOTES*

The empty string

SYMBOLS $\longrightarrow$ \time *LIST_QUOTES*

Any interval of time

LIST_SYMBOLS $\longrightarrow$ [*LIST_SYMBOLS_INTERN*]

Any of the symbols in the list

LIST_SYMBOLS $\longrightarrow$ [^ *LIST_SYMBOLS_INTERN*]

Any of the symbols *not* in the list

LIST_SYMBOLS $\longrightarrow$ [. . .]

Shortcut for "any symbol in the alphabet"

LIST_SYMBOL_INTERN $\longrightarrow$ *QUOTED_IDF*
$\longrightarrow$ *QUOTED_IDF LIST_SYMBOL_INTERN*
LIST_QUOTES $\longrightarrow$ ε
$\longrightarrow$ *LIST_QUOTES* '
OBRACKET ↑name $\longrightarrow$ $<$ { *IDF* ↑name }
OBRACKET ↑"\epsilon" $\longrightarrow$ $<$
CBRACKET ↑name $\longrightarrow$ {*IDF* ↑name } $>$ *INTERVAL*
CBRACKET ↑"\epsilon" $\longrightarrow$ $>$ *INTERVAL*

B.2.2 Concatenation and color brackets

The brackets do not have to be properly balanced, but any bracket opened must be closed once and only once after being opened, and any bracket closed must have been opened once and only once before being closed.

It is not possible for a couple of corresponding brackets to be at different levels of priority. The correspondence must be within the same list of concatenation.

Once a bracket is closed, it is possible to open it again like in

```
<{X} _a_ {X}>1 . <{X} _b_ {X}>2
```

When analyzing a list of concatenation, we keep in memory the list of non-closed opened brackets as an inherited attribute.

Keeping in mind that $\alpha \circ (\beta \cdot \gamma) \neq (\alpha \circ \beta) \cdot \gamma$, it appears necessary to avoid mixing standard and absorbing concatenation in a list containing colored brackets.

BRACKET_ATOM $\uparrow O, C$ $\longrightarrow$ *LIST_OBRACKET* $\uparrow O$
EXPRESSION
LIST_CBRACKET $\uparrow C$
LIST_CONCAT $\downarrow A$ $\longrightarrow$ *BRACKET_ATOM* $\uparrow O, C$
. *LIST_CONCAT* $\downarrow A + O - C$
condition : $C \subset A \cup O$
LIST_CONCAT $\downarrow A$ $\longrightarrow$ ε
condition : $A = \emptyset$
LIST_CONCAT_ABS $\downarrow A$ $\longrightarrow$ *BRACKET_ATOM* $\uparrow O, C$
o *LIST_CONCAT_ABS* $\downarrow A + O - C$
condition : $C \subset A \cup O$

"o" stands for absorbing concatenation.

LIST_CONCAT_ABS $\downarrow A$ $\longrightarrow$ ε
condition : $A = \emptyset$
LIST_OBRACKET $\uparrow A \cup \{\text{name}\}$ $\longrightarrow$ *OBRACKET* ↑name *LIST_OBRACKET* $\uparrow A$
LIST_OBRACKET $\uparrow\emptyset$ $\longrightarrow$ ε
LIST_CBRACKET $\uparrow A \cup \{\text{name}\}$ $\longrightarrow$ *CBRACKET* ↑name *LIST_CBRACKET* $\uparrow A$

$$LIST_CBRACKET \uparrow\emptyset \quad \longrightarrow \quad \varepsilon$$

B.2.3 Other operators

An ambiguous version of the grammar can be expressed as follow:

$$
\begin{array}{lcl}
EXPRESSION & \longrightarrow & SYMBOL \\
EXPRESSION & \longrightarrow & \texttt{(}\ EXPRESSION\ \texttt{)} \\
EXPRESSION & \longrightarrow & LIST_CONCAT \downarrow\emptyset \\
EXPRESSION & \longrightarrow & LIST_CONCAT_ABS \downarrow\emptyset \\
EXPRESSION & \longrightarrow & EXPRESSION \mid EXPRESSION \\
\quad \text{For } \varphi_1 \vee \varphi_2 & & \\
EXPRESSION & \longrightarrow & EXPRESSION\ \texttt{\^{}}\ EXPRESSION \\
\quad \text{For } \varphi_1 \wedge \varphi_2 & & \\
EXPRESSION & \longrightarrow & EXPRESSION\ \texttt{*} \\
\quad \text{For } \varphi^{*} & & \\
EXPRESSION & \longrightarrow & EXPRESSION\ \texttt{(*)} \\
\quad \text{For } \varphi^{\circledast} & & \\
EXPRESSION & \longrightarrow & EXPRESSION\ \texttt{+} \\
\quad \text{For } \varphi^{+} & & \\
EXPRESSION & \longrightarrow & EXPRESSION\ \texttt{(+)} \\
\quad \text{For } \varphi^{\oplus} & &
\end{array}
$$

B.2.4 Priority

The above definition is good from a theoretical point of view, but is deeply ambiguous. In the expression,

```
<{red} a* . b {red}>1
```

it is difficult to say whether the expression `<{red}` is attached to `a` (`(<{red} a)* . b {red}>1`) or to `a*` (`<{red} (a*) . b {red}>1`). In the first case, the expression is illegal (`<{red}` is never closed in `<{red}a`), and in the second, it is legal.

This is why the actual syntax is the following.

$$
\begin{array}{rcl}
EXPRESSION & \longrightarrow & EXPRESSION \mid EXPRESSION \\
 & \longrightarrow & EXPRESSION\ \texttt{\^{}}\ EXPRESSION \\
 & \longrightarrow & EXPRESSION\ \texttt{o}\ EXPRESSION \\
 & \longrightarrow & BRACKET_ATOM \\
BRACKET_ATOM \uparrow O, C & \longrightarrow & LIST_OBRACKET \uparrow O \\
 & & EXPRESSION_ACCEPTING_BRACKET \\
 & & LIST_CBRACKET \uparrow C \\
EXPRESSION_ACCEPTING_BRACKET & \longrightarrow & SIMPLE_EXPRESSION \\
 & \longrightarrow & EXPRESSION_ACCEPTING_BRACKET\ \texttt{*} \\
 & \longrightarrow & EXPRESSION_ACCEPTING_BRACKET\ \texttt{(o)} \\
 & \longrightarrow & EXPRESSION_ACCEPTING_BRACKET\ \texttt{+} \\
 & \longrightarrow & EXPRESSION_ACCEPTING_BRACKET\ \texttt{(+)} \\
SIMPLE_EXPRESSION & \longrightarrow & \texttt{(}\ EXPRESSION\ \texttt{)} \\
 & \longrightarrow & ATOM
\end{array}
$$

This gives a higher priority to the unary operators. Confusing notations such as

```
< a + * . b >1
```

Are no longer ambiguous. (Here, the expression is equivalent to `< ((a+)*) . b>`, as expected)

B.2.5 Examples

SYMBOLS

- A symbol alone:

  `SIMPLE_SYMBOL`

- A symbol with a quote, for renaming:

  `a'`

- Any delay followed by a symbol (because of the `_` around the identifier.)

  `_DELAYED_SYMBOL_`

- Just the empty string.

  `\epsilon`

- Any interval of time

  `\time`

  (note that `\time x` is equivalent to `_x_`)

- Either `a` or `b'`

  `[a b']`

- Anything but a `a` or a `b`.

  `[^ a b]`

- Any interval of time followed by any symbol in the alphabet (without quotes).

  `_[...]_`

LIST_CONCAT

- The simplest one

  `a . b`

- Bracket without concatenation (list of concatenation of length 1)

  `< _ONE_ >2`

- With brackets

  `< _a_ . <{Red} _b_* >[1, 12) . _c_ {Red}>3`

- Elements of the list can be expressions.

```
<{firstclock}
  a . _b_* .
  <{secondclock}
{firstclock}>(12, 125)
  . _c_
  {secondclock}>3
```

B.2.6 Declaration of the alphabet

The expressions defined above are actually encapsulated in a declaration, like the following example:

```
#alphabet ONE TWO THREE
#expression
ONE . _TWO_ . _TWO'_
```

It is illegal to declare twice the same symbol, and illegal to use a symbol if it has not been declared. Nevertheless, it is legal and sometimes necessary to declare a symbol without using it, because the alphabet also represents the synchronizing transitions of the future automaton.

FULL_EXPRESSION	⟶	`#alphabet` *ALPHABET_DECLARATION*
		`#expression` *EXPRESSION*
ALPHABET_DECLARATION	⟶	`IDF`
	⟶	*ALPHABET_DECLARATION* `IDF`

B.3 The Structure Of Abstract Regular Expression

B.3.1 Overview

The structure of timed regular expression is a tree: The type is an access to a record. The record contains some local information, and an array of pointers to the sons.

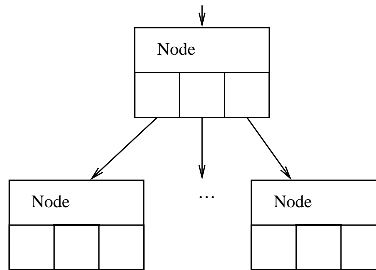

Figure B.2: The structure of tree

The record is parametrized by a type "node", and some local information may depend on the value of this discriminant.

The arity of each node and the allowed child at each position is also determined by the value of the node.

The constructors and mutators will do these verifications, to guarantee that the tree always corresponds to its grammar:

B.3.2 The tree grammar

full_expression	⟶	`REGEXP` (*alphabet*,*expression*)
alphabet	⟶	`LIST_SYMBOL`
expression	⟶	*symbols*
	⟶	*bracket_atom*
	⟶	*list_concat*
	⟶	*list_concat_abs*
	⟶	*concat*

Used for the binary concatenation before transforming it into list

	⟶	`N_OR` (*expression*, *expression*)
	⟶	`N_AND` (*expression*, *expression*)
	⟶	`STAR`(*expression*)
	⟶	`OSTAR`(*expression*)
	⟶	`PLUS`(*expression*)
	⟶	`OPLUS`(*expression*)
list_concat	⟶	`NODE_LIST_CONCAT` (*elem_list_concat*, *list_concat*)
	⟶	`EOL`

EOL for **E**nd **O**f **L**ist

elem_list_concat	⟶	`ELEM_LIST_CONCAT` (*list_obracket*, *expression*, *list_cbracket*)

LIST_OBRACKET is the list of brackets opening before the expression, and LIST_CBRACKET is the list of brackets closing after this one.

list_concat_abs	⟶	`NODE_LIST_CONCAT_ABS` (*elem_list_concat*, *list_concat_abs*)
	⟶	`EOL`
symbol	⟶	`SYMBOL` [`delay`, `list_symbol`, `complemented`]

`delay` is a boolean, `list_symbol` a set of symbol

list_obracket	⟶	`NODE_LIST_OBRACKET` (`OBRACKET`, *list_obracket*)
	⟶	`EOL`
list_cbracket	⟶	`NODE_LIST_CBRACKET` (`CBRACKET`, *list_bracket*)
	⟶	`EOL`
obracket	⟶	`OBRACKET`(`IDF`, `BRACKET_PAIR`[opening, closing])
cbracket	⟶	`CBRACKET` (`IDF`, `BRACKET_PAIR`[opening, closing], *bound*, *bound*)
bound	⟶	`BOUND` [`open`|`closed`, `value`]
bracket_atom	⟶	`BRACKET_ATOM` (*list_obracket*, *expression*, *list_cbracket*)
bracket_atom	⟶	`BRACKET_ATOM` (*list_obracket*, *list_notime*, *list_cbracket*)
list_notime	⟶	`NODE_LIST_NOTIME` (*expression*, *list_notime*)
	⟶	ε

B.3.3 Decoration

To enable the user of the structure to put additional information in the tree (This will be usefull for the construction of automata), a field is added to each node, pointing to an object of type `Decoration'Class`. To decorate the expression, one has to declare a derived type of Decoration, put whatever he or she wants in it, and use the functions `Set_Decoration` and `Get_Decoration` with it. Note that it is not necessary that all the decorations of an expression be of the same derived type.

This decoration is used during translation to store the associations of clocks and bracket. The opening bracket will be decorated and we'll be able to see this decoration from the matching closing bracket.

B.4 Parsing Expressions

Parsing regular expressions is done in two passes. The first one is context insensitive, and the second one is context sensitive to take care of color brackets.

B.4.1 Context Insensitive Analysis

The first pass is all the more common: The expressions are parsed as a usual expressions, operators are considered as unary or binary operators.

The tools AFLEX and AYACC, simmilar to FLEX and YACC are used.

The only difficulty is related to color brackets: In this pass, they are nearly ignored. When an expression like `<{red} x ...` is meet, the parser just fix the bracket "red" on the symbol "x". No paren matching is done.

B.4.2 Context Sensitive Analysis

This pass will do 3 main transformations:

- Transformation of binary concatenation into list,
- Moving color brackets and grouping expressions not accepting time.
- Color brackets matching.

It consists of one recursive infixed run on the tree. The recursive function returns a boolean saying if the expression accepts time.

When reaching a concatenation node, we enter in a mode of transformation of a list of concatenation. We initialize a structure of list of concatenation, and an empty "block" of expression which do not accept time elapse.

We continue the recursive run until a different concatenation or another operator is found. It is possible to go through a bracket atom if it contains the same concatenation as the one we are examining, as in `a . <(b . c)>1 . d`.

For each expression in the list, we add it to the list in different ways depending on the expression:

If the expression accepts time then, we add it as it is. If the last block of non time-accepting expressions is not empty, we add it to the list first.

If the expression does not accept time then, we add it to the last block of expressions not accepting time. Its brackets are added to the lists of brackets of the block (which is a bracket atom), and are considered common to all the block.

At the end of the list, we add the last block to the list.

Some more optimizations are done, including:

- When a symbol (say, `a`) is added to the last block, and if the last expression in the list is `\time`, then, we replace `\time` by `_a_`.
- When adding the last block to the list, the closing brackets are added to the last expression in the list.
- When adding an expression, the corresponding opening brackets are moved to the last block (to have all the bracket on the left as much as possible), except if the last block is empty.

The structure of the last block is a bracket atom, whose expression is a `LIST_NOTIME`.

B.5 The Structure Of Automaton

B.6 Generic automata

The structure of automata is designed to be as flexible as possible. In this program, I use only a narrowed class of timed automata, but many algorithms are the same, or similar for any kind of automata.

In the hope that my work could be reusable for other kind of automata, I used a mixture of generics and object oriented programming.

The template of automata is a generic package, taking as parameters two types, corresponding to the content of a state or of a transition.

The automata is then defined as shown in figure B.3

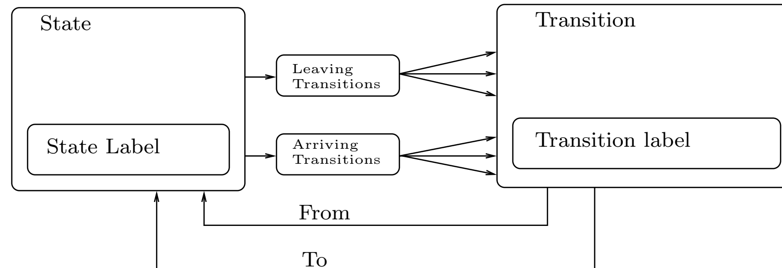

Figure B.3: The generic automata structure

The global structure of the automata being fixed, the main algorithms can be implemented at this level. However, changing the content of the states and transitions will change some algorithms. This is exactly the behavior of inheritance. This is why the generic automata is declared as a tagged record (Ada equivalent for class).

The implementation of a specific automata will then be an extension of the instantiation of the generic package. The instantiation gives the data to put in the automaton, and the inheritance enable redefining the primitives for manipulating the automaton. Once the generic automaton is written, the amount of work to define a new class of automaton should be minimal.

The basic operations (product, concatenation) are also generic procedure. For example, the concatenation takes as a generic argument a function to create the labels of the newly created transition. The use of generics allows a clear separation between the “time oriented” code and the generic code.

B.7 Timed automata

B.7.1 Instanciation of the generic automaton

In the version of timed automata used in this program, there is no invariant on states, and all the information is contained in the transitions.

The transitions contain a set of clock to reset, and a set of clock constraint.

Transitions and states contains a label “Comment” used only for the development: The manipulation of the automata can use the function `Add_Comment`, and the comments added will be printed as KRONOS comments when printing the automaton.

B.7.2 Clocks

Clocks are defined by a natural number. To write them to KRONOS format, a function `clock_name` is defined, returning simply the strings `x0`, `x1`...

A constraint on one clock is a pair *(clock, set of interval)* The set of interval should be seen as an union of intervals in which the clock value are allowed. Adding an interval to an union, in my implementation, optimizes the set of intervals. (No overlapping intervals)

B.8 Parsing Automata

B.9 Tests

The software has currently not been hardly tested, but provides some scripts and a small set of test cases to enable automatic and non-regression tests. About 60 test cases are provided.

There are currently 4 different types of tests:

test_print.sh Parses a regular expression, and prints it. The result should be similar to first one, but spacing has changed, parentheses has moved, ... That's why we do it a second time, and the result of the second compilation should allays be the same as the first. If not, the test fails.

test_error.sh Tries to parse an expression supposed to be incorrect. If the compilation succeeds, then, the test fails.

test_parse.sh This is a non regression test. When ran for the first time, it parses a file, displays the expression as a tree, and asks the user to confirm whether or not it is correct. If so, it archives the result. Next time the test is ran, the output is compared to the archive. If they do not differ, the test succeeds. Else, the `diff` of the outputs is shown to the user, and he has to confirm if the new output is still correct.

Note that this kind of test require the program to be deterministic, which is not the case when using hash sets with a hash function based on values of pointers.

test_translate.sh Same as `test_parse.sh`, but translates the expression into an automaton instead of just parsing it.

All these scripts run on a set of test cases.

A script called `test_all.sh` calls all these tests successively.

B.10 Practical Aspects

The language used for the project is Ada 95. It compiles with GNAT 3.14p, but not earlier versions which have a bug which make them incompatible with the Booch Components

The algorithmic program is encapsulated in scripts written in perl. Some other scripts useful for development and testing are written in the shell sh.

The Booch Components (Ada equivalent for C++'s STL) are used for the different sets, lists, and hash tables structures.

The parsers are written in Aflex and Ayacc.

For the BCG output, an Ada binding to the Bcg_user library, written in C, has been written.

The project use CVS as a version manager.

Compilation is managed by a Makefile.

The project has been developed on the GNU/Linux platform (on a 686 architecture). It should be portable to other unix platforms, but no effort have been made for portability outside the unix world.

B.11 Directory structure

The directory of the project is organized as follow:

Directory	Description
tests	Directory containing the tests
tests/tmp	Temporary directory to store the output of commands
tests/test_print_files	Test cases for test_print.sh, test_parse.sh and test_translate

Directory	Description
tests/test_error_files	Test cases for test_error.sh
tests/test_parse_archive	Archives for test_parse.sh
tests/test_translate_archive	Archives for test_translate.sh
Documentation	Documentation, report, ...
Documentation/presentation	Presentation done for TPTS in ETAPS 2002
Documentation/report	This report
Documentation/rapportDEA	My Master's Thesis
Scripts	Various kind of scripts usefull for the project.
examples	small case study
examples/train	The case of the train
src	Ada source files written for the project
src/bc	Add-on's to the booch components
src/bcg	Wrapper for the "BCG_User" library
src/common	Common to several parts
src/expressions	The structure of expressions
src/parseAutomata	Parser for automata
src/parseExpressions	Parser for expressions
src/tests	Small files used for testing and debugging
contrib	Source files used in the project, but not belonging to it
contrib/aflex-ayacc-for-gnat	Aflex and Ayacc parser generators
contrib/booch	The booch components
obj	Where the object files are stored
exec	Where the compiled executables are created

The environment variable `TREKHOME` should point to the root of the tree.

B.12 Programming Conventions

B.12.1 Defensive Programming

In most packages, two boolean variables are defined

- `defensive` must be `true` to enable the defensive code for the user of the package. Exceptions might be raised if the preconditions of the functions are not respected.
- `internal_defensive` must be `true` only for the development and debugging of the package. Once the package is stable, the defensive code should be useless for the user.

The exception `Internal_Error` is used only for that purpose. It should never be raised in a stable version.

B.12.2 Tracing

I often add debugging output to my programs. To avoid having to remove all those ugly `Put` and `Put_Line`, I introduce a small function called `Trace` in the body of packages. It adds the name of the package at the beginning of the line, and allows to decide the amount of information to display.

The variable `Trace_Level` tells how much should be displayed. 0 means no debug information, and 5 is the maximum.

B.12.3 Input/Output

Ada allows to use either `Standard_Output` or `Standard_Error` as a destination for output, and the default is `Standard_Error`.

The standard output should be limited to the actual output of the program (The resulting automaton or expression). Any debug information should go to the standard error.

Since it is important to avoid mixing standard output and standard error, a small package has been written: `limited_IO`. It only contains renaming of the some of the standard input/output functions: It allows the long forms like `Put(Standard_Error, "string");`, but does not allow the short forms like `Put("string");`.