

Automaton

PimenTech

SARL au capital de 51 000 francs

7bis, rue de Lesseps

75020 PARIS

<http://www.pimentech.net>

20 février 2002

1 Introduction

Le but de cette étude est de proposer une classe d'automates non déterministes (cf [HU79]), permettant de gérer efficacement l'ensemble des processus et leur enchaînements dans une infrastructure informatique.

2 Automate et dépendances

Tout processus Unix (cf [Rif93]) se termine en renvoyant un entier qui indique si tout s'est bien passé. La norme veut que l'entier 0 indique un fonctionnement normal et que tout autre valeur indique une erreur lors de son exécution. Ces codes d'erreurs sont notamment utilisés par de nombreux scripts shells dans les entreprises afin de recommencer des opérations s'étant mal déroulés ou encore d'avertir le responsable d'une défaillance dans un système informatique. On peut encore citer les fameuses *Makefiles* qui arrêtent l'enchaînement des opérations dès que l'une d'entre elles renvoie un code d'erreur différent de 0. Dans les *Makefiles*, chaque opération dépend d'une ou plusieurs autres. Une opération étant considérée ici comme la transformation d'un fichier d'un format vers un autre. Les *Makefiles* nous aident dans la gestion des dépendances. Le fichier transformé dépend de plusieurs fichiers sources, qui eux même peuvent dépendre d'autres fichiers, etc... Ainsi, une *Makefile* permet de mettre à jour un projet en ne reconstruisant que les fichiers qui dépendent des fichiers modifiés. La première utilisation des *Makefiles* est la compilation de programme constitués de plusieurs fichiers source, mais elles peuvent cependant être utilisés dans bien d'autres domaines où les temps de transformation des fichiers sont prohibitifs et où on ne souhaite pas tout régénérer à chaque fois (extraction de bases de données par exemple). Prenons maintenant un cas de figure où si une opération s'est mal déroulée il suffit de la relancer pour qu'elle réussisse au bout d'un nombre fini de fois (par exemple si l'erreur est due à une perte du réseau) (cf figure 1).

Les *Makefiles* ici ne nous sont d'aucun secours. Quant à la programmation directe d'un script gérant des dépendances même simples (cf figure 2), celui-ci deviendrait rapidement incompréhensibles (séries de `if then else` en cascade). Nous proposons donc des automates permettant de décrire proprement ce genre problèmes, et un algorithme les exécutant en gérant en parallèle les processus n'ayant aucune dépendance entre eux.

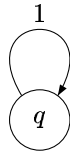


FIG. 1 – répétition d’une opération ayant échoué

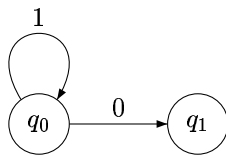


FIG. 2 – exécution de q_1 suite au succès de q_0

3 Dépendances et parallélisme

Le lancement en parallèle de plusieurs processus n’ayant aucune dépendance entre eux nécessite une gestion particulière pour les transitions en erreurs. Prenons l’exemple figure 3, ici q_1 et q_2 dépendent de l’exécution de q_0 cependant nous remarquons que lors d’un échec de l’un ou l’autre nous avons une transition vers q_0 or l’un comme l’autre ont été exécutés en parallèle. Ainsi, les transitions en erreurs provoquent une terminaison des processus dépendant de l’état sur lequel on arrive. Ici, si q_1 se termine en erreur, puisque nous avons une transition en erreur vers q_0 les processus dépendant de q_0 seront détruits, soit ici, q_2 . En effet q_3 n’a pas été exécuté puisqu’il dépend de q_1 et q_2 .

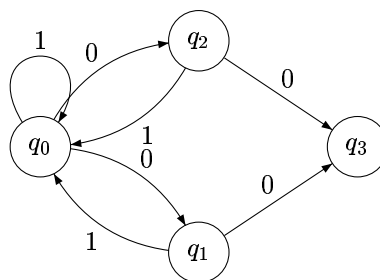


FIG. 3 – exécution de q_1 et q_2 en parallèle.

4 Algorithme

Nous donnons ici l’algorithme utilisé, mais nous n’y décrivons pas le parallélisme qui est une adaptation de la partie 4 dans la fonction *exec*.

Soit \mathcal{C} l'ensemble des automates que nous considérons. Soit $\mathcal{A} = (Q, A, E, q_0) \in \mathcal{C}$ avec l'alphabet $A = \mathbb{N}$. Soit le vecteur $V = (V_{q_0}, V_{q_1}, \dots, V_{q_{|Q|-1}})$ avec $V_{q_i} \subset Q = \emptyset$, représentant l'exécution au temps t . Soit le vecteur $W = (W_{q_0}, W_{q_1}, \dots, W_{q_{|Q|-1}})$ avec $W_{q_i} = \{q \in Q / (q, 0, q_i) \in E\}$. Soit le vecteur $W^t = (W_{q_0}^t, W_{q_1}^t, \dots, W_{q_{|Q|-1}}^t)$ avec :

$$W_{q_i}^t = \begin{cases} W_{q_i} & \text{if } i > 0, \\ \emptyset & \text{otherwise.} \end{cases}$$

function $exec(\mathcal{A} \in \mathcal{C}, q \in Q) \in \mathbb{N}$:

1. **let** $n = exec_state(q) \in \mathbb{N}$.
2. **let** $N = next(q, n) = \{q' \in Q / (q, n, q') \in E\}$.
3. **let** $V_q = \emptyset$.
4. **for each** $q' \in N$:
 - (a) **if** $n = 0$,
 - then** // on success
 - i. $W_{q'}^t = W_{q'}^t - \{q\}$.
 - ii. **if** $W_{q'}^t = \emptyset$,
 - then** // dependencies ok
 - A. $kill(q')$. // clean up
 - B. $V_q = V_q \cup \{q'\}$.
 - C. $exec(\mathcal{A}, q')$ &.
 - else** // on error
 - i. $kill(q')$. // abort all
 - ii. $V_q = V_q \cup \{q'\}$.
 - iii. $exec(\mathcal{A}, q')$ &.
5. **return** 0.

function $kill(q \in Q) \in \mathbb{N}$:

1. $W_q^t = W_q$.
2. **for each** $q' \in V_q$:
 - (a) $V_q = V_q - \{q'\}$.
 - (b) $kill(q')$.
3. $kill_state(q)$.
4. **return** 0.

5 Implémentation

Cet algorithme a été implémenté en C++ (cf [Str96]), et peut se télécharger sur notre site <http://www.pimentech.net/pimentech/site/technologies/ressources>. Nous utilisons entre autres les bibliothèques *Xerces* (cf <http://xml.apache.org/xerces-c>) pour parser le *XML* ainsi que *STL* (cf [SL95]) pour créer nos types graphes.

5.1 Syntaxe

La description de l'automate se fait en *XML* :

Exemple 1 *Cet exemple correspond à l'automate de la figure 4.*

```
<AUTOMATON>

<STATE name='start' command='sh'>
echo starting
</STATE>

<STATE name='end' command='sh'>
echo ending
</STATE>

<STATE name='q1' command='sh'>
echo q1 starting
sleep 1
echo q1 finish
</STATE>

<STATE name='q2' command='sh'>
echo q2 starting
sleep 2
echo q2 finish
</STATE>

<TRANSITION from='start' label='0' to='q1' />
<TRANSITION from='start' label='0' to='q2' />
<TRANSITION from='q1' label='0' to='end' />
<TRANSITION from='q2' label='0' to='end' />

</AUTOMATON>
```

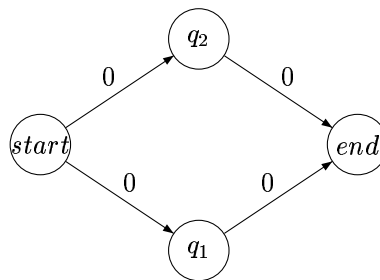


FIG. 4 – exécution de q_1 et q_2 en parallèle.

L'exécution de cet automate fournit le résultat suivant :

```
ramon@cumin:~/src/automaton/bin/examples$ automa-
ton waiting.xml
starting
q2 starting
q1 starting
```

```
q1 finish
q2 finish
ending
ramon@cumin:~/src/automaton/bin/examples$
```

On peut y remarquer l'exécution de q_1 et q_2 en parallèle ainsi que l'attente de end jusqu'à ce que q_1 et q_2 se soient terminés avec succès.

Le programme *automaton* impose que l'état de départ s'appelle *start*. De plus, tout ce qui se trouve entre les tags XML `<STATE>` est envoyé sur l'entrée standard de la commande donnée en argument de la variable `command`.

Exemple 2 *Cet exemple correspond à l'automate de la figure 5*

```
<AUTOMATON>

<STATE name='start' command='sh' >
echo starting
</STATE>

<STATE name='end' command='sh' >
echo ending
</STATE>

<STATE name='q1' command='sh' >
echo q1 starting
sleep 1
echo q1 finished on error
exit 1
</STATE>

<STATE name='q2' command='sh' >
echo q2 starting
sleep 3
echo q2 finish
</STATE>

<TRANSITION from='start' label='0' to='q1' />
<TRANSITION from='q1' label='1' to='start' />
<TRANSITION from='start' label='0' to='q2' />
<TRANSITION from='q1' label='0' to='end' />
<TRANSITION from='q2' label='0' to='end' />

</AUTOMATON>
```

L'exécution de cet automate fournit le résultat suivant :

```
ramon@cumin:~/src/automaton/bin/examples$ automaton waiting-
error.xml
starting
q1 starting
q2 starting
q1 finished on error
starting
q2 starting
q1 starting
q2 starting
```

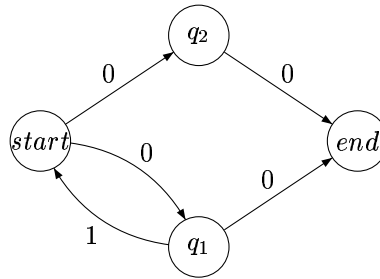


FIG. 5 – exécution de q_1 et q_2 en parallèle avec échec de q_1 .

```

q1 finished on error
starting
q2 starting
q1 starting
q2 starting
q1 finished on error
starting
q2 starting
q1 starting
q2 starting
q1 finished on error
...
..
.
<Ctrl-C>
ramon@cumin:~/src/automaton/bin/examples$
  
```

6 Utilisation

6.1 Domaines d'application

Automaton est un outil qui permet de séparer la gestion des processus de la gestion des relations entre ces processus. La description des automates se fait en XML, donc en format texte ne requérant aucun logiciel spécifique. De ce point de vue, n'importe quelle commande capable de retourner un code de statut peut constituer un processus. Les domaines d'application d'un tel superviseur d'automates sont donc très variés, de la programmation de procédures d'administration système au contrôle de chaîne de production.

Les utilisations qui en ont été réalisées jusqu'à présent, à des fins expérimentales, incluent :

- migration de données
- surveillance des défaillance d'une connexion réseau lors de transferts de données
- réplication de données pour des SGBD ne gérant pas ce type de comportement en natif

6.2 Statut du développement

L'élaboration d'*automaton* réclame toutefois encore de nombreuses études de cas concrets car nous avons découvert qu'il y a plusieurs algorithmes possibles pour poursuivre l'exécution d'un automate en fonction des statut de ces processus ; il va de soit que

l'utilisation concrète que l'on souhaite faire de l'automate influence ce choix.

Il pourrait également être intéressant pour certains automates de rajouter directement dans *automaton* la gestion de timeout (délais d'exécution maximum pour chaque processus) pour les processus ne pouvant pas gérer eux même cet évènement, ou bien de gérer un système d'appel pour coordonner et éventuellement interrompre des processus (par exemple pour des tâches s'exécutant sur des systèmes distincts qui doivent être synchronisés – dans ce cas, *automaton* considérerait d'abord certaines variables du système, comme l'horloge, avant de considérer le statut retourné par un processus).

Automaton devrait aussi proposer un certain nombre de méthodes pour reconnaître les dead-lock de l'automate (détection des états de l'automate qui ne permettraient plus à l'automate de cheminer vers un ensemble d'états-solution).

Il faut également tester ce qui se passe lorsqu'*automaton* est stoppé puis relancé : certains processus peuvent d'eux même savoir qu'ils n'ont pas à être relancés, mais on peut aussi mettre en place une variable dans la description XML de l'automate qui fournit une équivalence à l'exécution d'un processus (par exemple, par analogie aux *Makefiles*, que tel fichier est à jour) afin d'alléger la description de l'automate et réduire son nombre d'état. Ceci fait, on pourrait alors mettre en place un système qui vérifie l'exécution d'*automaton* lui même (temps d'exécution, dead-lock détecté au vol, ressources consommées – notamment nombres de process lancés) et qui le relance en cas de dépassement des assertions.

Enfin, toujours dans le but de simplifier la description des automates, *automaton* pourrait de lui même gérer plusieurs prétendants par tâche, pour exécuter des processus différents lorsqu'une tâche est déclarée inopérante (par exemple, remplacement d'une connection scp par une connection ftp en cas de défaillance répétée de la connection scp).

Toutes ces améliorations possibles ainsi que le test exhaustif du comportement d'*automaton* en conditions de production requièrent encore de nombreux tests avant d'envisager la mise en production réelle du programme.

Suite à quoi *automaton* pourrait remplacer avantageusement de nombreuses *makefile* et *cron-scripts* (programme gérant des processus sur une base horaire) pour superviser de nombreux aspects des systèmes informatiques.

Références

- [HU79] J.E. Hopcroft and J.D. Ullmann. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.
- [Rif93] J.M. Riflet. *La programmation sous Unix*. Ediscience International, 1993.
- [SL95] A. Stepanov and M. Lee. The standard template library. Technical report, Hewlett-Packard Laboratories, 1995.
- [Str96] B. Stroustrup. *Le Langage C++*. International Thomson Publishing France, 1996.