Hes·so VALAIS WALLIS
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences Western Switzerland

Domaine Sciences de l'ingénieur
Rte du Rawyl 47
CH- 1950 Sion 2
Tél. +41 27 606 85 11
Fax +41 27 606 85 75
info@hevs.ch

www.hevs.ch

Filière Systèmes industriels

Orientation Infotronics

Diplôme 2010

Jean Iwanowski

SCEdit

Professeur Medard Rieder

Expert Johannes Scheier

Sion, le 12 juillet 2010

HES-SO Valais

SI	TV
X	X

Données du travail de diplôme
Daten der Diplomarbeit

FO.1.2.02.07.AB
pof/31/01/2009

☒ FSI ☐ FTV	Année académique / *Studienjahr* **2009/10**	No TD / *Nr. DA* **it/2010/25**
Mandant / *Auftraggeber* ☒ HES—SO Valais ☐ Industrie ☐ Etablissement partenaire	Etudiant / *Student* **Jean Iwanowski**	Lieu d'exécution / *Ausführungsort* ☒ HES—SO Valais ☐ Industrie ☐ Etablissement partenaire
Professeur / *Dozent* **Medard Rieder**	Expert / *Experte* *(données complètes)*	
Travail confidentiel / *vertrauliche Arbeit* ☐ oui / ja [1] ☒ non / nein		

Titre / *Titel*

SCEdit

Description et Objectifs / *Beschreibung und Ziele*

In a model based tool chain for embedded system programming, code generation and execution framework (XF) play an important role. Such a tool chain has partially been developed by Hugo Nunes, a former student. In order to validate his work and in order to optimize the existing tool chain during the present bachelor project, the following tasks have to be accomplished:

- Understand the existing XF. Optimize and complete the interface of the existing XF
- Implement XF for an embedded target in C and a QT target
- Develop C-code generation and C++ code generation out of a XMI model containing a simple state machine
- Develop SCXML generation out of a XMI model containing a simple state machine
- Test everything. Report on test results
- Documentation: For each step, technical documentation has to be established. Also, a final report has to be written. The technically important features have to be outlined in this report.

Délais / *Termine*

Attribution du thème / *Ausgabe des Auftrags:*
22.02.2010

Remise du rapport / *Abgabe des Schlussberichts:*
12.07.2010, 12:00

Remise du rapport intermédiaire / *Zwischenbericht:*
07.05.2010, 17:00

Exposition publique / *Ausstellung Diplomarbeiten:*
27.08.2010

Défense intermédiaire / *Zwischenverteidigung:*
21.05.2010

Défense orale / *Mündliche Verteidigung:*
Semaine 35 / *Woche 35*

Signature ou visa / *Unterschrift oder Visum*

Responsable de la filière
Leiter des Studiengangs: .....

[1] Etudiant/*Student*: .....

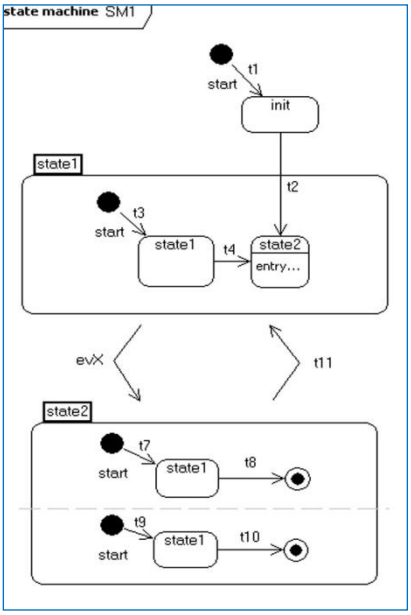
---

[1] *Par sa signature, l'étudiant-e s'engage à respecter strictement la directive et le caractère confidentiel du travail de diplôme qui lui est confié et des informations mises à sa disposition.*
Durch seine Unterschrift verpflichtet sich der Student, die Richtlinie einzuhalten sowie die Vertraulichkeit der Diplomarbeit und der dafür zur Verfügung gestellten Informationen zu wahren.

Rapport reçu le / *Schlussbericht erhalten am* ..... Visa du secrétariat / *Visum des Sekretariats* .....

Hes·so VALAIS WALLIS
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences Western Switzerland

Titre du travail
50 caractères au maximum



Diplômant/e Jean Iwanowski

Objectif du projet

Le projet SCEdit vise à développer une chaine d'outils capable de transformer un modèle de machine d'états-transitions UML, stocké au format XMI, en code exécutable pour la cible logicielle Qt.

Méthodes | Expériences | Résultats

Le projet se décompose en deux parties distinctes, relatives aux deux outils à développer. Le premier est un eXecution Framework (XF). Celui-ci définit une interface standard fournissant les services nécessaires à l'exécution d'un modèle qui contient des machines d'états-transitions sur la cible choisie. Comme il est difficile de déterminer à priori si Qt peut remplir seul toutes les tâches d'un XF, celui-ci est développé en parallèle à Qt, dont il exploite certaines possibilités.
Le second outil est la génératrice de code, qui transforme un fichier XMI en code C++ implémentant des machines d'états-transitions et exploitant les services fournis par le XF.
Le développement du XF est mené à bien. Celui-ci est capable d'exécuter des machines d'état simples. Ce travail montre que Qt est capable d'assumer toutes les tâches assumées par le XF développé.
Le développement d'une version de travail de la génératrice de code est réalisé. Perfectible, cette génératrice permet néanmoins d'exécuter des machines d'états-transitions complexes, au moyen du State Machine Framework proposé par Qt. Qt est utilisé comme XF en lieu et place de celui développé initialement pour le projet.

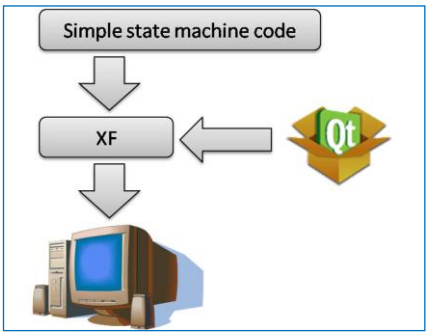
Travail de diplôme
| édition 2010 |

Filière
Systèmes industriels

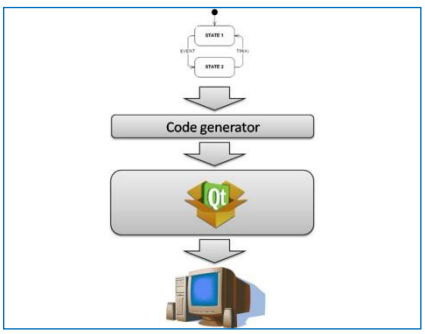
Domaine d'application
Infotronics

Professeur responsable
M. Medard Rieder
Medard.Rieder@hevs.ch

Partenaire
-



Première approche :
le XF développé et Qt travaillent de concert pour permettre l'exécution d'une machine d'états simple codée à la main.



Seconde approche :
La génératrice transforme un modèle XMI en code. Qt est seul responsable de fournir les services nécessaires à son exécution.

HES-SO Valais
Route du Rawyl 47
1950 Sion

Tél. 027 606 85 11
URL www.hevs.ch

1. Table des matières

1. TABLE DES MATIÈRES 1

2. TABLE DES FIGURES 4

3. TABLE DES EXTRAITS DE CODE 4

4. PRÉAMBULE 5

4.1. REMARQUE CONCERNANT LES EXTRAITS DE CODE 5

5. SPÉCIFICATION 6

5.1. DESCRIPTIF DU PROJET 6
5.2. CAHIER DES CHARGES 8
5.2.1. Cahier des charges initial 8
5.2.2. Cahier des charges redéfini 8

6. OUTILS 9

6.1. ECLIPSE VERSION : 3.4.1 9
6.2. TOPCASED VERSION : 2.3.0 9
6.3. MDWORKBENCH VERSION : 3.0.0 9
6.4. QT SDK 9
6.4.1. Qt Creator Version : 1.3.1 9
6.4.1. Qt Library Version 4.6.3 10

7. QT EXECUTION FRAMEWORK (QXF) 11

7.1. STRUCTURE 11
7.2. MÉCANISMES 12
7.2.1. Traitement des événements 12
7.2.2. Traitement des timers 13
7.3. IMPLÉMENTATION 13
7.3.1. Généralités 13
7.3.2. Classe "QxfReactive" 14
7.3.3. Classe "QxfEventDispatcher" 14
7.3.4. Classe "QxfEventQueue" 15
7.3.5. Classe "QxfEvent" 15
7.3.6. Classe "QxfStartBehaviorEvent" 15
7.3.7. Classe "QxfDefaultEvent" 15
7.3.8. Classe "QxfTimeoutEvent" 15
7.3.9. Classe "QxfTimeoutManager" 16
7.3.10. Classe "QxfTimeoutList" 16
7.3.11. Classe "QxfFactory" 16
7.3.12. Implémentation des machines d'états-transitions 16
7.3.13. Différenciation des événements 17
7.4. TESTS 18
7.5. LIMITATIONS DU QXF 20
7.6. RÉORIENTATION DU PROJET 20

8. UTILISATION DE DE QT COMME XF 21

8.1. IMPLÉMENTATION DES MACHINES D'ÉTAT 21
8.1.1. Machine d'etats-transitions 21
8.1.2. Etat 21
8.1.3. Etat imbriqué 21
8.1.4. Etat initial 21
8.1.5. Etat final 21
8.1.6. Transition 22
8.1.7. Régions 22
8.1.8. Sous-machine 22
8.2. MÉCANISMES 22
8.2.1. Traitement des événements 22
8.2.2. Traitement des timers 22
8.3. STRUCTURE 23
8.4. TESTS 23

9. GÉNÉRATRICE DE CODE 24

9.1. MODEL QUERY LANGUAGE (MQL) 24
9.2. TEXT GENERATION LANGUAGE (TGL) 24
9.3. IMPLÉMENTATION DE LA GÉNÉRATRICE 25
9.3.1. Ficher « Generator.mqr » 25
9.3.2. Fichier « Main.tgt » 26
9.3.3. Fichier « Project.tgt » 26
9.3.4. Fichiers « Factory » 26
9.3.5. Fichier « StateMachineCommon.tgt » 26
9.3.6. Fichiers « StateMachine» 26
9.3.7. Fichiers « State» 27
9.3.8. Fichiers « Region » 27
9.3.9. Fichiers « Transition » 27
9.3.10. Fichier « uml21_Association.mqs » 27
9.3.11. Fichier « uml21_Class.mqs » 28
9.3.12. Fichier « uml21_Element.mqs » 28
9.3.13. Fichier « uml21_InstanceSpecification.mqs » 28
9.3.14. Fichier « uml21_Behavior.mqs » 29
9.3.15. Fichier « uml21_Region.mqs » 29
9.3.16. Fichier « uml21_State.mqs » 30
9.3.17. Fichier « uml21_StateMachine.mqs » 30
9.3.18. Fichier « uml21_Transition.mqs » 30

10. UTILISATION DE LA CHAÎNE D'OUTILS 32

10.1. PROCÉDURE 32
10.2. CRÉATION D'UN MODÈLE UML 32
10.2.1. Eléments de la norme UML utilisés 32
10.2.2. Diagramme de classes 33
10.2.3. Diagrammes de machines d'états-transitions 34
10.2.4. Ajout des triggers 35

10.2.5. Ajout des guards, et des actions d'entrée, sortie, transition 36
10.2.6. Validation du modèle 37
10.3. TESTS 37
10.4. ERREURS CONNUES 38
11. CONCLUSION 39
12. DÉVELOPPEMENTS FUTURS 39
13. REMERCIEMENTS 40
14. BIBLIOGRAPHIE 41

2. Table des figures

FIGURE 1 : ADAPTATION D'UN MODÈLE À UNE CIBLE 6
FIGURE 2 : SERVICES D'UN XF 7
FIGURE 3 : CHAINE D'OUTILS 7
FIGURE 4 : STRUCTURE DU XF 11
FIGURE 5 : TRAITEMENT DES ÉVÉNEMENTS 12
FIGURE 6 : TRAITEMENT DES TIMERS 13
FIGURE 7 : CLASSES DÉVELOPPÉES 14
FIGURE 8 : MACHINE D'ÉTATS-TRANSITIONS DE TEST 18
FIGURE 9 : RÉSULTAT DE TEST DU QXF 19
FIGURE 10 : SÉQUENCE DES OPÉRATIONS 32
FIGURE 11 : DIAGRAMME DE CLASSES 33
FIGURE 12 : DIAGRAMME DE MACHINES D'ÉTATS-TRANSITIONS 34
FIGURE 13 : AJOUT DES TRIGGERS 35
FIGURE 14 : AJOUT DES GUARDS ET ACTIONS 36
FIGURE 15 : PREMIÈRE MACHINE DE TEST 37
FIGURE 16 : DEUXIÈME MODÈLE DE TEST 38

3. Table des extraits de code

CODE 1 : ENVOI D'ÉVÉNEMENTS 17
CODE 2 : MÉTHODE TYPEOF 18
CODE 3 : EXEMPLE DE CODE MQL 24
CODE 4 : EXEMPLE DE TEMPLATE TGL 25
CODE 5 : TEXTE GÉNÉRÉ PAR LE TEMPLATE TGL 25

4. Préambule

4.1. Remarque concernant les extraits de code

Afin de faciliter leur compréhension, les extraits de code présentés dans ce document sont épurés de toutes les lignes qui ne participent pas à l'illustration de l'explication associée. Ils ne sont donc pas totalement similaires au code développé dans le cadre de ce projet. Les annexes contenant le code complet sont disponibles en annexe.

5. Spécification

5.1. Descriptif du projet

Ce projet vise à assembler une chaine d'outils capable de transformer un modèle UML en code exécutable pour une cible donnée. Cette cible peut être matérielle (microcontrôleur, processeur,…) ou logicielle. La cible choisie ici est Qt, outil très répandu et apprécié notamment pour ses capacités dans le domaine des interfaces utilisateur.

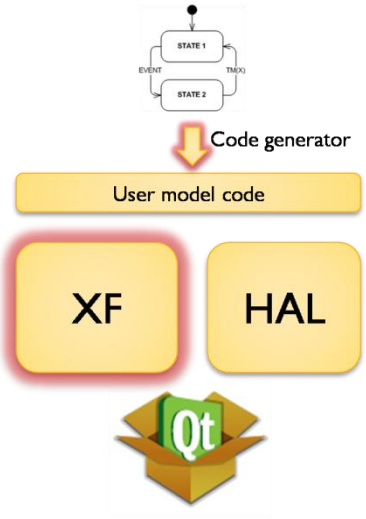


Figure 1 : Adaptation d'un modèle à une cible

La Figure 1 schématise le processus d'adaptation : le modèle doit tout d'abord être traduit en code, dans un langage compréhensible par la cible. C'est le rôle de la génératrice de code.

Afin de masquer les spécificités de chaque cible, celle-ci est accédée par le code généré au travers d'une couche d'abstraction. Cette couche est divisée en deux composants : le Hardware Abstraction Layer (HAL) interface le matériel associé à la cible. Dans le cas de Qt, l'accès direct au matériel n'est pas permis, raison pour laquelle le HAL n'est pas traité plus avant. L'eXecution Framework (XF) interface les services standards que toute cible doit fournir :

- Gestion des événements
- Gestion des timers
- Gestion de la mémoire
- Gestion des processus
- Gestion des sections critiques

Les services du XF sont représentés sur la Figure 2. Il faut remarquer ici que la cible logicielle Qt posséde certaines caractéristiques d'un XF. Il est cependant difficile de déterminer à priori si Qt est capable d'assumer toutes les fonctions d'un XF.

Figure 2 : Services d'un XF

Dans le doute, décision est prise de développer un XF spécifique. Celui-ci fera néanmoins appel à certaines fonctionnalités de Qt.

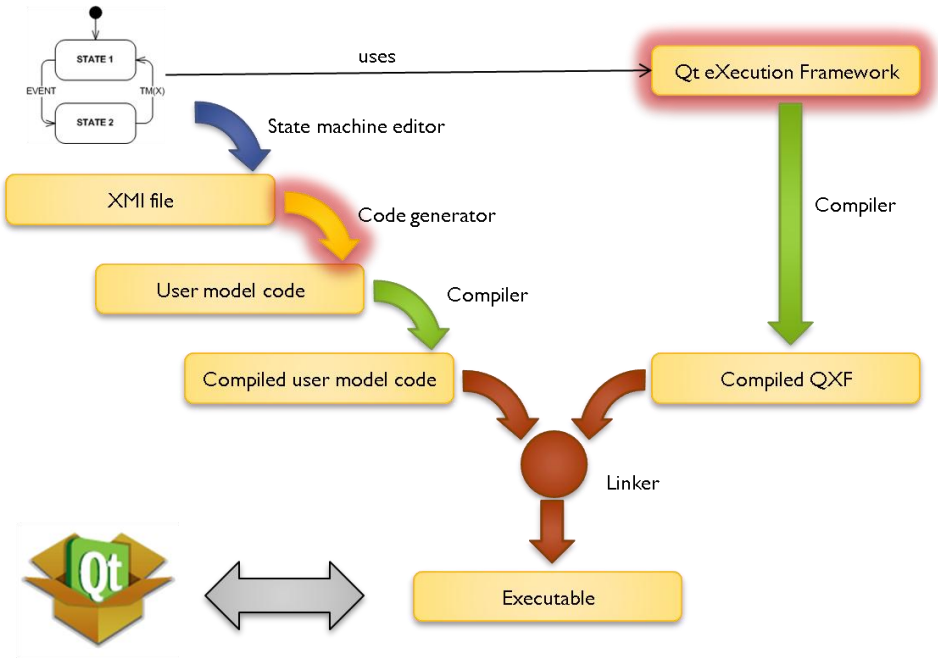


Figure 3 : Chaine d'outils

Le XF et la génératrice de code développés dans le cadre de ce projet s'intègrent dans une chaine d'outils, représentée sur la Figure 3. Celle-ci est capable de transformer en fichier exécutable Qt le modèle dessiné par l'utilisateur, sans presque aucune intervention de sa part.

Les outils préexistants utilisés dans la chaine sont décrits au §6.

5.2. Cahier des charges

5.2.1. Cahier des charges initial

Le cahier des charges du projet tel qu'initialement défini est le suivant :

- Etudier le fonctionnement du XF développé dans le cadre d'un travail de diplôme antérieur (Nunes, 2008).
- Compléter l'interface du XF susmentionné.
- Implémenter un XF en langage C pour une cible embarquée, en C++ pour la cible QT.
- Développer des génératrices de code interprétant le modèle XMI d'une machine d'états simple pour les XF susmentionnés.
- Développer une génératrice de code SCXML à partir du modèle XMI d'une machine d'états simple.
- Tester les outils développés.

5.2.2. Cahier des charges redéfini

Afin de tenir compte des contraintes temporelles du projet ainsi que de certaines difficultés techniques, le cahier des charges du projet a été redéfini comme suit, d'un commun accord avec les professeurs et assistants responsables :

- Etudier le fonctionnement du XF développé dans le cadre d'un travail de diplôme antérieur (Nunes, 2008).
- Implémenter un XF en C++ pour la cible QT.
- Développer une génératrice de code interprétant le modèle XMI d'une machine d'états simple pour le XF susmentionné.
- Tester les outils développés.

6. Outils

6.1. Eclipse Version : 3.4.1

Eclipse est un environnement de développement extensible open-source. Il permet l'utilisation des plugins Topcased et MDWorkbench.

6.2. Topcased Version : 2.3.0

Topcased est un environnement logiciel dédié à la réalisation de systèmes embarqués critiques. Fruit du travail commun de divers partenaires (Airbus, Thales et bien d'autres), il est distribué sous la forme d'un plugin open-source gratuit pour l'environnement de développement Eclipse.

Topcased est utilisé dans ce projet pour ses capacités de modélisation graphique ainsi que pour son respect de la norme UML. Il a pour rôle de générer le fichier XMI représentant le modèle, à partir des informations entrées par l'utilisateur.

6.3. MDWorkbench Version : 3.0.0

MDWorkbench est un plugin Eclipse très performant pour la génération de code et la transformation de modèles. Il permet d'effectuer de nombreuses opérations sur les modèles par le biais du Model Query Language (MQL), et de générer des fichiers texte par le biais du Text Generation Language (TGL).

6.4. Qt SDK

Qt est un framework multiplateformes pour applications et interfaces utilisateur. Il contient une librairie de classes, des outils et un environnement de développement intégrés. Parmi ces éléments on peut mentionner :

6.4.1. Qt Creator Version : 1.3.1

Qt Creator est un environnement de développement pour C++ intégrant la librairie Qt. Il offre un accès simple à la documentation de Qt, une complétion de code performante, un compilateur et un debugger.

6.4.1. Qt Library Version 4.6.3

La librairie Qt fournit des classes simplifiant la réalisation de tâches de haut niveau, comme la gestion d'interfaces utilisateur. Elle fournit également un framework relatifs aux machines d'état particulièrement intéressant dans le cadre de ce projet.

7. Qt eXecution Framework (QXF)

7.1. Structure

Le QXF doit pouvoir gérer événements et timers dans un environnement multi processus. Pour ce faire la structure présentée sur la Figure 4.

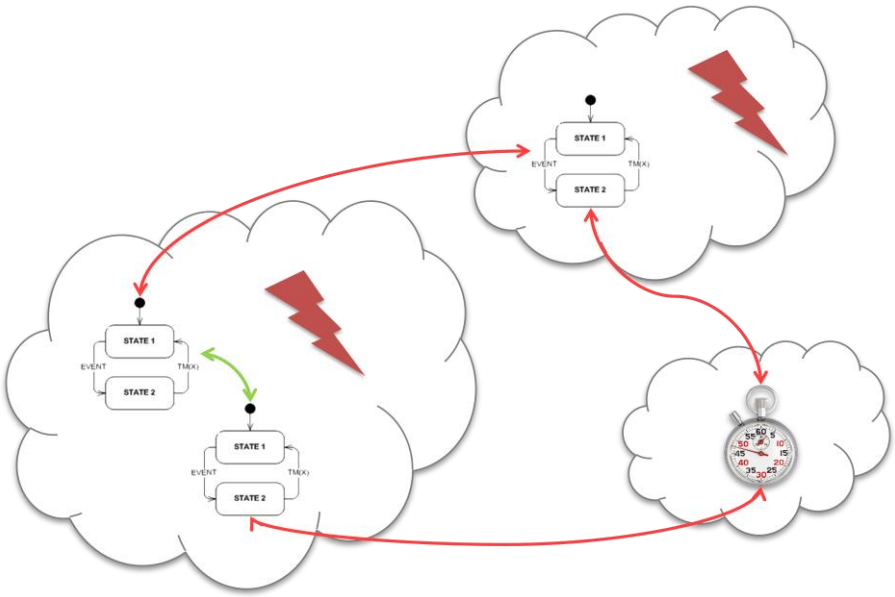


Figure 4 : Structure du XF

Chaque processus (symbole : nuages) contient une ou plusieurs machines d'états-transitions. Afin que celles-ci puissent communiquer entre elles, chaque processus est également équipé d'un distributeur d'événements (symbole : éclairs). Celui-ci est chargé de sérialiser les événements envoyés au processus et de les distribuer aux machines d'états-transitions affiliées à ce dernier.

Pour que les communications entre processus s'effectuent de manière thread-safe, la transmission d'événements est assurée par le mécanisme signal-slot de Qt. Celui-ci permet de découpler totalement les deux parties impliquées dans la communication. Pour plus d'informations à ce sujet, se référer à la documentation de Qt (Nokia Corporation, 2010).

La gestion des timers (symbole : chronomètre) est confiée à un processus dédié de priorité maximale, de sorte à minimiser l'influence des autres composants du projet sur la gestion du temps, et donc à optimiser sa précision.

7.2. Mécanismes

7.2.1. Traitement des événements

Le traitement des événements est schématisé sur la Figure 5.

L'événement créé par l'expéditeur est envoyé à la machine d'états-transitions par le mécanisme signal-slot. Lorsque la machine d'états-transition reçoit l'événement, elle le passe au distributeur d'événements de son processus, qui le place dans une file d'attente.

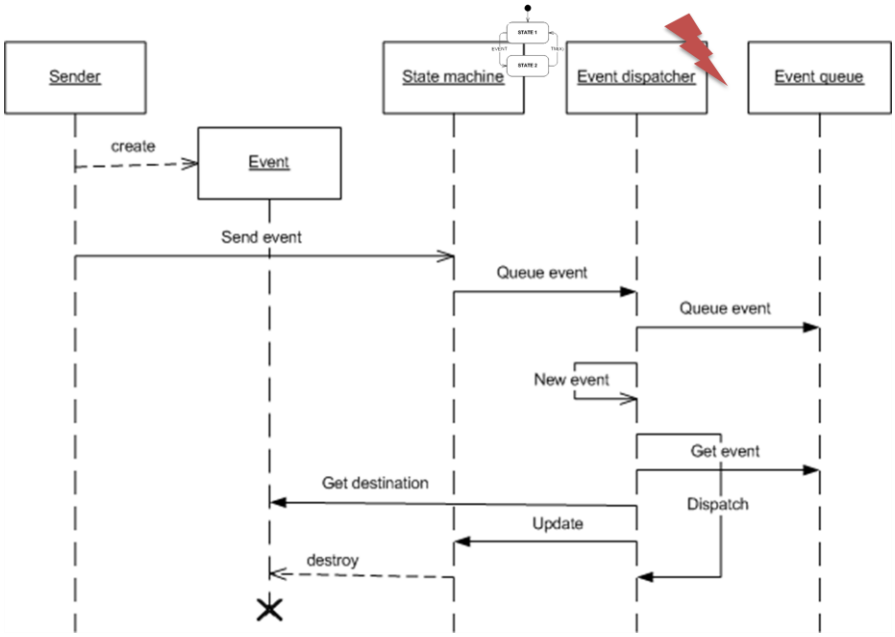


Figure 5 : Traitement des événements

Les boucles de scrutation sont couteuses en ressources. Afin d'éviter que le distributeur d'événements se renseigne sur l'état de la file d'attente par ce biais, le mécanisme signal-slot est de nouveau mis à contribution : à chaque fois que le distributeur place un événement dans la file, il s'envoie un signal à lui-même. Ce signal à pour effet de déclencher aussitôt que possible la distribution de l'événement.

Chaque instance de la classe « Evénement » a pour attribut l'adresse de la machine d'états-transition à laquelle elle est destinée. Ainsi le distributeur peut en prendre connaissance et transmettre l'événement à son destinataire. Enfin la machine d'états-transitions réagit en fonction de l'événement reçu.

7.2.2. Traitement des timers

Le traitement des timers est schématisé sur la Figure 6.

Le timer est un type particulier d'événement. Outre l'adresse de son initiateur, il a pour attribut la quantité de temps le séparant de son échéance. Lorsqu'il en a besoin, l'expéditeur transmet un timer au gestionnaire de timers. Celui-ci le stocke dans une liste, et le décrémente au rythme d'un signal d'horloge défini au préalable.

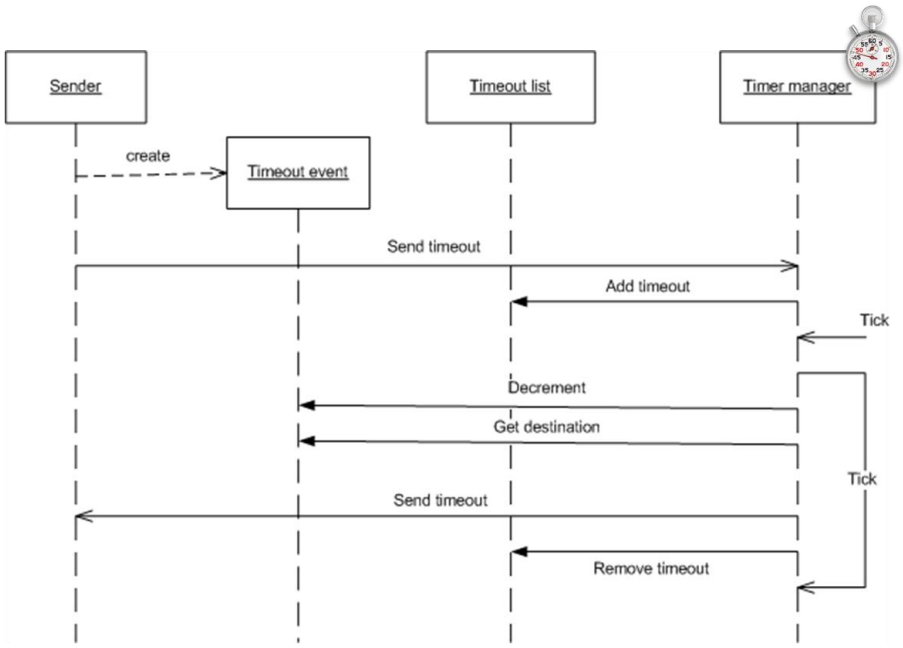


Figure 6 : Traitement des timers

Une fois le timer arrivé à échéance, le gestionnaire de timers le renvoie à son émetteur, de la même manière que celle décrite plus haut pour les événements.

7.3. Implémentation

7.3.1. Généralités

Le diagramme de classes du QXF est présenté à la Figure 7. Cette section décrit le rôle de chacune des classes du QXF. Pour des informations plus détaillées quant à l'implémentation de chaque classe, se référer à la documentation du QXF à l'annexe 1, ou au code de chacune de ses classes figurant également en annexe.

7.3.2. Classe "QxfReactive"

Cette classe représente une abstraction des machines d'états-transitions telles qu'implémentées dans le QXF. Toutes les machines d'états-transitions spécifiques à un modèle particulier doivent en conséquence être dérivées de cette classe.

Cette classe comporte plusieurs méthodes virtuelles pures, qui doivent être redéfinies par les machines d'états-transitions dérivées :

- « typeOf » détermine le type des événements entrants
- « behavior » définit la réaction de la machine à un événement entrant
- « initRelations » initialise les liens vers d'autres machines

Elle contient également un pointeur sur le distributeur d'événements (« QxfEventDispatcher ») du processus auquel elle est affiliée.

Le code de cette classe figure sur l'annexe 2.

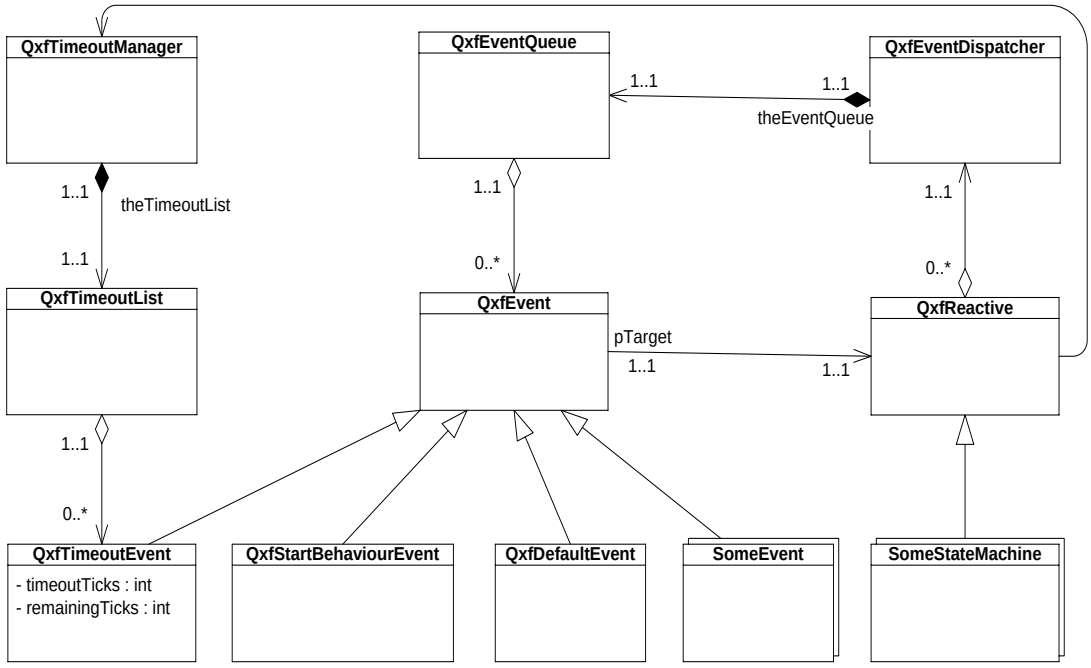


Figure 7 : Classes développées

7.3.3. Classe "QxfEventDispatcher"

Cette classe est charge de stocker puis de distribuer les événements aux machines d'états-transitions du processus dans lequel elle réside. Pour ce faire elle dispose d'une

queue (« QxfEventQueue ») dans laquelle elle stocke les messages avant de les redistribuer.

Le code de cette classe figure sur l'annexe 3.

7.3.4. Classe "QxfEventQueue"

Cette classe permet le stockage des événements avant leur redistribution. Elle spécialise la classe « QQueue » de Qt pour ne permettre que le stockage de pointeurs sur des événements (« QxfEvent »). Cette spécialisation se fait par le biais d'un typedef.

Le code de cette classe figure sur l'annexe 4.

7.3.5. Classe "QxfEvent"

Cette classe est une abstraction des événements tels qu'implémentés dans le QXF. Elle a pour attribut la destination de l'événement. Tous les événements spécifiques à un modèle particulier doivent être dérivés de cette classe.

Le code de cette classe figure sur l'annexe 5.

7.3.6. Classe "QxfStartBehaviorEvent"

Cette classe est une spécialisation de la classe « QxfEvent » interne au QXF. Des instances de cette classe sont utilisées pour démarrer les machines d'états-transitions.

Le code de cette classe figure sur l'annexe 6.

7.3.7. Classe "QxfDefaultEvent"

Cette classe est une spécialisation de la classe « QxfEvent » interne au QXF. Des instances de cette classe sont utilisées pour déclencher les transitions par défaut des machines d'états-transitions.

Le code de cette classe figure sur l'annexe 7.

7.3.8. Classe "QxfTimeoutEvent"

Cette classe est une spécialisation de la classe « QxfEvent » interne au QXF. Elle définit des attributs supplémentaires permettant au gestionnaire de timers de connaître le temps restant avant l'échéance du timer.

Des instances de cette classe sont utilisées pour déclencher les transitions par défaut des machines d'états-transitions.

Le code de cette classe figure sur l'annexe 8.

7.3.9. Classe "QxfTimeoutManager"

Cette classe est instanciée une seule fois dans le QXF. Elle doit impérativement avoir un processus dédié, car elle gère l'aspect temporel de l'exécution des machines d'états-transitions.

Elle stocke les « QxfTimeoutEvent » que lui envoient les machines d'état dans sa liste de timers (« QxfTimeoutList »). Pour décrémenter les compteurs des timers, elle dispose d'une instance de la classe « Qtimer » de Qt, qui est capable d'émettre un signal à intervalles réguliers.

Le code de cette classe figure sur l'annexe 9.

7.3.10. Classe "QxfTimeoutList"

Cette classe permet de traiter les "QxfTimeoutEvent" envoyés par les machines d'état. Afin de minimiser le coût en ressources de l'opération de décrémentation, ces timers sont stockés sous forme d'une liste chainée de timers. Cette technique permet de ne décrémenter qu'un compteur, les valeurs figurant dans les timers chainés étant relatives à l'échéance du précédent. Pour plus d'informations à ce sujet, se référer au document « Execution Framework » (Rieder & Steiner, Execution Framework, 2009)

Le code de cette classe figure sur l'annexe 10.

7.3.11. Classe "QxfFactory"

Cette classe est responsable de l'instanciation de tous les objets nécessaires au bon fonctionnement du QXF. Elle est elle-même instanciée dans la fonction « main » de l'application Qt.

Le code de cette classe figure sur l'annexe 11.

7.3.12. Implémentation des machines d'états-transitions

L'implémentation des machines d'états-transitions dans le QXF s'inspire d'un pattern utilisé dans le cadre de la HES-SO // Valais pour la programmation de cibles embarquées. Ce pattern assimile les états et transitions aux valeurs d'énumérations y relatives. Chaque

événement déclenche la mise à jour de variables qui mémorisent l'ancien état, l'état futur et la transition actuelle. Les actions de sortie, de transition et d'entrée appropriées, contenues dans des « switch-case », sont effectuées en fonction du contenu de ces variables. Pour plus d'informations au sujet de ce pattern, se référer au document « Machines d'états-transitions et l'implémentation des machines d'états-transitions » (Rieder & Steiner, 2009)

7.3.13. Différenciation des événements

Afin d'exploiter les possibilités offertes par l'orientation objet, la méthode de différenciation des événements par un identificateur numérique n'a pas été retenue. L'option choisie est celle de créer une nouvelle classe dérivée de « QxfEvent » pour chaque type d'événement.

Afin de pouvoir envoyer n'importe quel événement au moyen d'une même fonction malgré leur caractère hétérogène, on exploite le mécanisme des templates. La seule fonction « sendEvent » figurant sur le Code 1 permet ainsi d'envoyer n'importe quel type d'événement.

```
template<typename T>
void createEvent(QxfReactive* pTarget)
{
   // creates a connexion with target
    connect(
            this,
            SIGNAL( sendEvent( QxfEvent* ) ),
            pTarget,
            SLOT( receiveEvent( QxfEvent* ) ),
            Qt::QueuedConnection
            );
    // creates the event to be sent
    QxfEvent* pEvent = new T (pTarget);
    // sends the event
    emit sendEvent(pEvent);
    // ends the connextion with target
    disconnect(
            this,
            SIGNAL( sendEvent( QxfEvent* ) ),
            pTarget,
            SLOT( receiveEvent( QxfEvent* ) )
            );
}
```

Code 1 : envoi d'événements

Les machines d'états-transitions doivent également pouvoir déterminer le type d'un événement lors de sa réception. Pour ce faire le pointeur sur une instance de la classe « QxfEvent » à examiner est soumis à un dynamic_cast vers des pointeurs sur tous les types d'événements existants. Seule la conversion vers un pointeur sur le type effectif de

l'événement retourne un résultat valide, renseignant ainsi l'utilisateur sur sa nature. Ceci permet de créer une méthode « typeOf », dont un exemple figure sur le Code 2.

```
int SimpleStateMachine2::typeOf(QxfEvent* pEvent)
{
    if( dynamic_cast<QxfStartBehaviorEvent*>( pEvent ) )
    {
        return EV_START_BEHAVIOR;
    }
    else if( dynamic_cast<QxfTimeoutEvent*>( pEvent ) )
    {
        return EV_TIMEOUT;
    }
    else if( dynamic_cast<QxfDefaultEvent*>( pEvent ) )
    {
        return EV_DEFAULT;
    }
    else if( dynamic_cast<TestEvent*>( pEvent ) )
    {
        return EV_TEST;
    }
    else return -1;
}
```

Code 2 : Méthode typeOf

7.4. Tests

Afin de tester le QXF, deux machines d'état simples sont implémentées, selon le schéma de la Figure 8. Ces deux machines sont affiliées à des processus différents. Le code correspondant figure sur l'annexe 12.

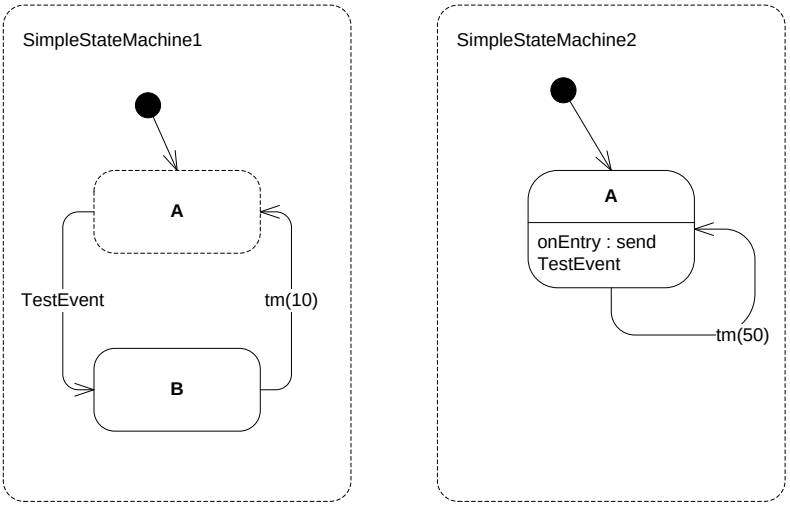


Figure 8 : Machine d'états-transitions de test

La Figure 9 montre la sortie du QXF exécutant ces machines d'états-transitions, depuis le démarrage des machines jusqu'à ce que le système qu'elles composent ait retrouvé son état initial. On peut en tirer les constats suivants :

- L'enchainement des états est correct. Le comportement séquentiel des machines d'état est donc valide.
- L'entrelacement des outputs des deux machines d'état confirme que chacune évolue bel et bien dans un processus indépendant.
- La gestion des timers chainés est fonctionnelle.

```
Enter : SM1 A
Enter : SM2 A
----------------------
Add timeout :
Timeout 50 , remaining = 50
----------------------
----------------------
Send TestEvent
----------------------
Enter : SM1 B
----------------------
Timeout 50 , remaining = 44
Add timeout :
Timeout 10 , remaining = 10
Timeout 50 , remaining = 34
----------------------
----------------------
Timeout reached
Timeout 50 , remaining = 34
----------------------
Enter : SM1 A
----------------------
Timeout reached
----------------------
Enter : SM2 A
```

Figure 9 : Résultat de test du QXF

Pour des raisons de temps, le comportement temporel du système n'est pas testé avec précision, une telle mesure requérant en effet une longue préparation. On constate néanmoins que le rapport entre les durées des deux timers utilisés semble se retrouver lors de l'exécution des machines d'état.

Le QXF devrait également être testé avec d'autres machines d'état, afin de valider dans des conditions différentes. Cette idée est abandonnée en raison du temps important de rédaction d'une machine d'états-transition ainsi que des considérations mentionnées au point suivant.

7.5. Limitations du QXF

Le QXF permet donc l'exécution de machines d'états-transitions simples. Il serait toutefois appréciable de pouvoir exécuter des machines plus complexes, comprenant des sous-machines, des régions… Or implémenter ce genre de machines d'états-transitions selon le pattern choisi est très complexe, et le temps à disposition pour ce travail n'est pas suffisant pour mener cette tâche à bien.

7.6. Réorientation du projet

Pour dépasser cet écueil, décision est prise de choisir un nouveau pattern orienté objets, dans lequel les divers éléments de la norme UML relative aux machines d'états-transitions sont représentés par des classes. Or Qt dispose d'un Framework dédié aux machines d'états-transitions. Des informations relatives à ce framework sont disponibles dans la documentation de Qt (Nokia Corporation, 2010). Etant donné que l'adaptation en cours de réalisation est effectuée pour la cible Qt, ce framework semble un candidat parfait pour l'implémentation des machines d'état.

L'étude du State Machine Framework montre qu'il est capable de fournir tous les services proposés par le QXF. Celui-ci peut donc être abandonné, et ses fonctionnalités assumées par Qt.

8. Utilisation de de Qt comme XF

8.1. Implémentation des machines d'état

8.1.1. Machine d'etats-transitions

Une machine d'états-transitions est implémentée par une instance d'une classe dérivée de la classe « QStateMachine ». Cette spécialisation est indispensable, du fait que chaque machine d'état a une structure unique, et doit donc pouvoir bénéficier d'un constructeur spécifique.

8.1.2. Etat

Un état est implémenté par une instance d'une classe dérivée de la classe « QState ». Il est nécessaire de créer une nouvelle classe dérivée pour chaque nouvel état. En effet la classe « QState » contient les méthodes « onEntry » et « onExit », qui sont automatiquement appelées à l'entrée respectivement la sortie de l'état. Chaque état ayant des actions d'entrée et de sortie uniques, il faut donc pouvoir ré-implémenter ces méthodes pour chacun d'entre eux.

8.1.3. Etat imbriqué

Un état imbriqué est implémenté en étant défini comme enfant de l'état ou de la machine d'états-transition qui le contient, selon le principe des arbres d'objets de Qt. Pour plus d'informations au sujet de ce mécanisme, se référer à la documentation de Qt (Nokia Corporation, 2010).

8.1.4. Etat initial

Un état initial ne peut être implémenté au sens strict. Dans le State Machine Framework de Qt cet élément n'existe pas. Sa fonctionnalité peut cependant être reproduite en définissant le premier état suivant l'état initial comme point d'entrée de son parent. Ceci est réalisé par le biais de la fonction « setInitialState » de son parent.

8.1.5. Etat final

Un état final est implémenté par une instance de la classe « QFinalState ». Cette classe n'a pas besoin d'être dérivée, étant donné qu'un état final ne peut avoir d'action d'entrée ou de sortie.

8.1.6. Transition

Une transition est implémentée par une instance d'une classe dérivée de la classe « QSignalTransition ». Comme pour les états, la spécialisation est nécessaire afin de différencier les actions de transition des diverses transitions (méthode « onTransition »).

8.1.7. Régions

Les régions sont implémentées sous la forme d'états, dont l'attribut « childMode » reçoit la valeur « ParallelStates ».

8.1.8. Sous-machine

Une sous-machine est implémentée comme une machine d'états-transitions standard. Comme pour les états imbriqués, elle est ensuite définie comme enfant de l'état qui la contient.

8.2. Mécanismes

8.2.1. Traitement des événements

Les transitions dérivées de la classe « QSignalTransition » peuvent être déclenchées par un signal Qt. Ce procédé présente le double avantage d'être extrêmement simple et d'assurer le découplage entre les processus impliqués.

Pour instancier une telle transition il faut donner en paramètre à son constructeur le nom du signal requis et l'adresse de son émetteur. Ceci pose un problème, dans la mesure où l'événement peut être émis par n'importe qui.

Pour résoudre ce problème, on confie l'émission du signal à la machine d'états-transitions contenant la transition, par le biais d'une méthode publique. Ainsi l'émetteur du signal est connu lors de l'instanciation de la transition.

Lorsque l'événement déclencheur de la transition doit être émis, il incombe à l'émetteur d'appeler la méthode appropriée de la machine d'états-transitions de destination.

8.2.2. Traitement des timers

Les transitions temporisées sont gérées de la même manière que les transitions déclenchées par événements, le signal déclenchant la transition étant cette fois le signal « timeout » d'une instance de la classe « QTimer ».

Il faut toutefois tenir compte du fait qu'un timer est démarré par un état particulier et ne concerne que ce même état. S'il arrive à échéance, il ne peut déclencher qu'une transition sortante de cet état. Si une autre transition est déclenchée, il doit par contre être interrompu. L'émission des signaux « timeout » ne peut donc cette fois être centralisée au niveau de la machine d'états-transition, sous peine ne pouvoir déterminer l'initiateur du timer.

Ce problème est résolu en attribuant un « QTimer » à chaque état qui possède une transition temporisée. Le timer est ensuite démarré en première instruction de la méthode « onEntry » de l'état, et arrêté en première instruction de la méthode « onExit ».

8.3. Structure

En raison de l'implémentation décrite ci-dessus, le nombre de classes nécessaire au fonctionnement d'une machine d'états-transitions est conséquent. Or l'utilisateur de ce code – qui sera généré à partir d'un modèle – n'a pas besoin d'avoir connaissance de toutes ces classes, une machine d'états fonctionnant de manière parfaitement autonome.

En conséquence, décision est prise d'adopter la structure suivante :

- Chaque classe représentant une machine d'états-transitions est stockée dans un fichier d'en-têtes et un fichier de définitions à son nom.
- Chaque classe constitutive d'une machine d'états-transitions est stockée dans les fichiers d'en-têtes respectivement de définitions de sa classe parente
- Une Classe « Factory » est responsable de l'instanciation des machines d'états-transitions, de leur liaison avec les autres machines, de leur démarrage et de leur affiliation à leurs processus dédiés.

Cette structure permet de présenter à l'utilisateur un nombre de fichiers réduit et un point d'accès unique à toutes les classes générées.

8.4. Tests

Etant donné qu'une machine d'états-transitions complexe est nécessaire pour tester le fonctionnement de Qt en tant que XF, les tests seront effectués une fois la génératrice de code réalisée. Ainsi sera possible d'éviter le long travail d'écriture de cette machine d'états.

9. Génératrice de code

9.1. Model Query Language (MQL)

Le langage MQL permet d'effectuer des requêtes sur un modèle. Sa syntaxe est simplifiée au maximum, de sorte à ce que le développeur puisse porter toute son attention à la sémantique des requètes.

Dans ce projet, il est principalement utilisé pour sélectionner certains éléments du modèle, ainsi que pour lancer l'exécution de templates TGL, mais ses capacités vont bien au dela de cet emploi. Le MQL peut notamment effectuer des opérations de transformation sur un modèle.

```
package com.mycompany.example;

// expects a loaded UML 2.1 model as input
public ruleset GenerateAllJavaClasses(in model : uml21) {

    public rule generate() {

        // loop on each Class of the UML 2.1 model
        foreach (class in model.getInstances("Class")) {

            // calls the text template GenerateJavaClass
            $GenerateJavaClass(class);
        }
    }
}
```

Code 3 : Exemple de code MQL

Plus d'informations sur le MQL sont disponibles dans l'aide de MDWorkbench, dont est tiré l'exemple du Code 3. Le code de cet exemple a pour effet d'appeler le template « GenerateJavaClass » pour chaque classe faisant partie du modèle passé en paramètre.

9.2. Text Generation Language (TGL)

Le TGL est un langage qui permet de définir des templates pour la génération de fichiers texte. Ces templates sont composés de texte statique, et de sections dynamiques. Ces dernières sont délimités par des balises, qui pemettent de générer le texte en fonctions des données extraites d'un modèle. La sélection des données du modèle peut se faire au moyen de requêtes MQL. Le Code 4 présente un exemple de template tiré de l'aide de Topcased.

```
[#package com.mycompany.example]

[#-- we define a template GenerateJavaFile which expects a UML class argument -
-]
```

```
[#template GenerateJavaFile(class : uml21.Class)]
[#file]${class.name}.java[/#file] [#-- the file where to write generated
contents --]
public class ${class.name} {

    public String toString() {
        return "${class.name} instance";
    }
}
[/#template]
```

Code 4 : Exemple de template TGL

Le résultat de l'exécution de ce template donne le texte présenté sur le Code 5.

```
public class Account {

    public String toString() {
        return "Account instance";
    }
}
```

Code 5 : Texte généré par le template TGL

Ce langage est très simple d'utilisation. Une documentation détaillée des possibilités qu'il offre est disponible dans l'aide de MDWorkbench.

9.3. Implémentation de la génératrice

9.3.1. Ficher « Generator.mqr »

Ce fichier est le point d'entrée de la génératrice. Il prend en paramètre le modèle soumis à la génératrice, et appelle les templates TGL nécessaires à la génération du projet, à savoir :

- « Main.tgt »
- « Project.tgt »
- « FactoryHeader.tgt »
- « FactoryBody.tgt »
- « FactoryBody.tgt »
- « FactoryBody.tgt »
- « StateMachineCommon.tgt »
- « StateMachineHeader.tgt » pour chaque classe du modèle contenant une machine d'états-transition
- « StateMachineBody.tgt » pour chaque classe du modèle contenant une machine d'états-transition

Le code de ce fichier figure sur l'annexe 13.

9.3.2. Fichier « Main.tgt »

Ce fichier génère la fonction main de l'application, dont la seule fonction est de créer une instance de la Factory du projet, puis de lancer la boucle principale de Qt. Son contenu est purement statique. Il n'est utile qu'à des fins de debug : dans des conditions normales, l'utilisateur instancie lui-même la Factory dans le fichier main de l'application qu'il développe.

Le code de ce fichier figure sur l'annexe 14.

9.3.3. Fichier « Project.tgt »

Ce fichier génère le fichier .pro qui permet à Qt Creator de connaître tous les fichiers impliqués dans le projet. Il n'est également utile qu'à des fins de débug, puisque dans des conditions d'utilisation normales l'utilisateur génère lui-même ce fichier.

Le code de ce fichier figure sur l'annexe 15.

9.3.4. Fichiers « Factory »

Les fichiers « FactoryHeader.tgt » et « FactoryBody.tgt » génèrent les fichiers d'en-têtes et de définitions de la Factory du projet.

Le code de ces fichiers figure sur l'annexe 16.

9.3.5. Fichier « StateMachineCommon.tgt »

Ce fichier génère le fichier d'en-têtes « StateMachineCommon.h », dont le but est de fournir à toutes les machines la macro permettant l'envoi d'événements. Son contenu est purement statique.

Le code de ces fichiers figure sur l'annexe 17.

9.3.6. Fichiers « StateMachine»

Les fichiers « StateMachineHeader.tgt » et « StateMachineBody.tgt » génèrent les fichiers d'en-têtes et de définitions de chaque machine d'états-transitions du projet. Comme expliqué au § 8.3, ces fichiers doivent non seulement contenir les classes représentant les machines d'états-transitions, mais également celles représentant les états et transitions

contenues dans ces machines. C'est pourquoi les fichers « StateMachineHeader.tgt » et « StateMachineBody.tgt » appellent pour chacun de ces éléments les templates suivants :

- « StateHeader.tgt » respectivement « StateBody.tgt »
- « RegionHeader.tgt » respectivement « RegionBody.tgt »
- « TransitionHeader.tgt » respectivement « TransitionBody.tgt »

Le code de ces fichiers figure sur l'annexe 18.

9.3.7. Fichiers « State»

Les fichiers « StateHeader.tgt » et « StateBody.tgt » génèrent le code des déclarations et des définitions des classes représentant les états d'une machine d'états-transitions.

Le code de ces fichiers figure sur l'annexe 19.

9.3.8. Fichiers « Region »

Les fichiers « RegionHeader.tgt » et « RegionBody.tgt » génèrent le code des déclarations et des définitions des classes représentant les régions concurrentes imbriquées dans un état. Comme mentionné au § 8.1.7, ces régions doivent être implémentées dans Qt sous forme d'états.

Le code de ces fichiers figure sur l'annexe 20.

9.3.9. Fichiers « Transition »

Les fichiers « TransitionHeader.tgt » et « TransitionBody.tgt » génèrent le code des déclarations et des définitions des classes des transitions d'une machine d'états-transitions.

Le code de ces fichiers figure sur l'annexe 21.

9.3.10. Fichier « uml21_Association.mqs »

Ce fichier contient les scripts suivants :

- *"getClassesRelatedTo(class : uml21.Class) : MDWList"*
  Ce script s'applique à une association. Il retourne une liste de toutes les classes liées à la classe passée en paramètre par l'association pour laquelle le script est exécuté.
  Ce script est utile pour déterminer le nombre et le type des pointeurs à déclarer

dans chaque classe représentant une machine d'états-transitions et pointant vers d'autres machines.

Le code de ce fichier figure sur l'annexe 22.

9.3.11. Fichier « uml21_Class.mqs »

- *"hasStateMachine() : Boolean"*
  Ce script s'applique à une classe. Il retourne « true » si la classe pour laquelle le script est exécuté possède un comportement du type machine d'états-transitions. Ce script est utile pour déterminer quelles classes doivent être soumises à la génératrice de code.

Le code de ce fichier figure sur l'annexe 23.

9.3.12. Fichier « uml21_Element.mqs »

- *"getClass() : uml21.Class"*
  Ce script s'applique à n'importe quel élément du modèle. Il remonte dans l'arborescence du modèle afin de retrouver la classe qui contient l'élément pour lequel le script est exécuté.
- *"getParent() : uml21.Element"*
  Ce script s'applique à n'importe quell element du modèle. Il remonte dans l'arborescence du modèle pour trouver le parent de l'élément pour lequel le script est execute. Parent s'entend ici au sens des machines d'états-transitions de Qt. Il peut donc s'agir soit d'une machine d'états-transitions, soit d'un état, soit d'une region concurrente.
  Ce script est utile pour déterminer la filiation à attribuer à un état Qt lors de son instanciation.
- *"pathName() : String"*
  Ce script s'applique à n'importe quel élément du modèle. Il remonte dans l'arborescence du modèle jusqu'à la classe qui contient l'élément pour lequel le script est exécuté. Chaque parent rencontré – au sens des machines d'états-transitions Qt – voit son nom concaténé à une chaine de caractères. Celle-ci constitue la valeur de retour du script.
  Ce script est utile pour attribuer à chaque classe Qt un nom unique, relatif à sa position dans l'arborescence du modèle.

Le code de ce fichier figure sur l'annexe 24.

9.3.13. Fichier « uml21_InstanceSpecification.mqs »

- *"hasStateMachine() : boolean"*
  Ce script s'applique à une instance. Il retourne « true » si le classifier de l'instance pour laquelle le script est exécuté possède une machine d'états-transitions.
  Ce script est utile pour déterminer quelles instances représentent une machine d'états.
- *"isActive() : Boolean"*
  Ce script s'applique à une instance. Il retourne « true » si le classifier de l'instance pour laquelle le script est exécuté est défini comme actif.
  Ce script est utile pour déterminer quelles classes doivent être affiliées à un processus dédié.
- *"isAssociation() : Boolean"*
  Ce script s'applique à une instance. Il retourne « true » si le classifier de l'instance pour laquelle le script est exécuté est une association.
  Ce script est utile pour différencier les instances de classes des instances d'associations.
- *"isClass() : Boolean"*
  Ce script s'applique à une instance. Il retourne « true » si le classifier de l'instance pour laquelle le script est exécuté est une classe.
  Ce script est utile pour différencier les instances de classes des instances d'associations.

Le code de ce fichier figure sur l'annexe 25.

9.3.14. Fichier « uml21_Behavior.mqs »

- *"concat() : String"*
  Ce script s'applique à un « opaque behavior ». Il retourne une chaîne de caractères résultant de la concaténation de toutes les lignes de code contenues dans le « body » du comportement pour lequel le script est exécuté.
  Ce script est utile pour récupérer toutes les lignes de code introduites dans le modèle par l'utilisateur, et les insérer dans le code généré.

Le code de ce fichier figure sur l'annexe 26.

9.3.15. Fichier « uml21_Region.mqs »

- *"isConcurrent() : Boolean"*
  Ce script s'applique à une région. Il retourne « true » s'il existe au moins une région concurrente à la région pour laquelle le script est éxécuté.

Le code de ce fichier figure sur l'annexe 27.

9.3.16. Fichier « uml21_State.mqs »

- *"getTimedTransition() : int"*
  Ce script s'applique à un état. Il recherche l'éventuelle transition déclenchée par un timer sortant de l'état pour lequel le script est exécuté. Si celle-ci existe, le script retourne la valeur entière associée.
  Ce script est utile pour trouver la valeur à assigner au timer Qt d'un état.
- *"isInitial() : Boolean"*
  Ce script s'applique à un état. Il retourne « true » si l'état pour lequel il est exécuté possède une transition entrante qui provient d'un état initial.
  Ce script est utile pour détecter les états qui doivent être définis dans Qt comme états initiaux.

Le code de ce fichier figure sur l'annexe 28.

9.3.17. Fichier « uml21_StateMachine.mqs »

- *"isOrthogonal() : Boolean"*
  Ce script s'applique à une machine d'états-transitions. Il retourne « true » si la machine pour laquelle le script est exécuté contient plusieurs régions concurrentes.
  Ce script est utile pour déterminer si des états parallèles doivent être rajoutés dans Qt comme enfants de la machine, afin de simuler le comportement des régions concurrentes.

Le code de ce fichier figure sur l'annexe 29.

9.3.18. Fichier « uml21_Transition.mqs »

- *"isInitial() : Boolean"*
  Ce script s'applique à une transition. Il retourne « true » si la transition pour laquelle le script est exécuté provient d'un état initial.
- *"isSignalTriggered() : Boolean"*
  Ce script s'applique à une transition. Il retourne « true » si la transition pour laquelle le script est déclenchée par un événement de type signal.
- *"isTimeTriggered () : Boolean"*
  Ce script s'applique à une transition. Il retourne « true » si la transition pour laquelle le script est exécuté est déclenchée par un événement de type temporel.
- *"isTriggered() : Boolean"*
  Ce script s'applique à une transition. Il retourne « true » si la transition pour laquelle le script est exécuté possède un déclencheur.

Le code de ce fichier figure sur l'annexe 30.

10. Utilisation de la chaîne d'outils

10.1. Procédure

L'utilisation des outils Topcased, MDWorkbench et QtCreator n'est pas abordée ici en détail. Pour plus d'informations, se référer aux nombreux tutoriels disponibles sur internet. Un résumé des étapes à suivre se trouve à la Figure 10.

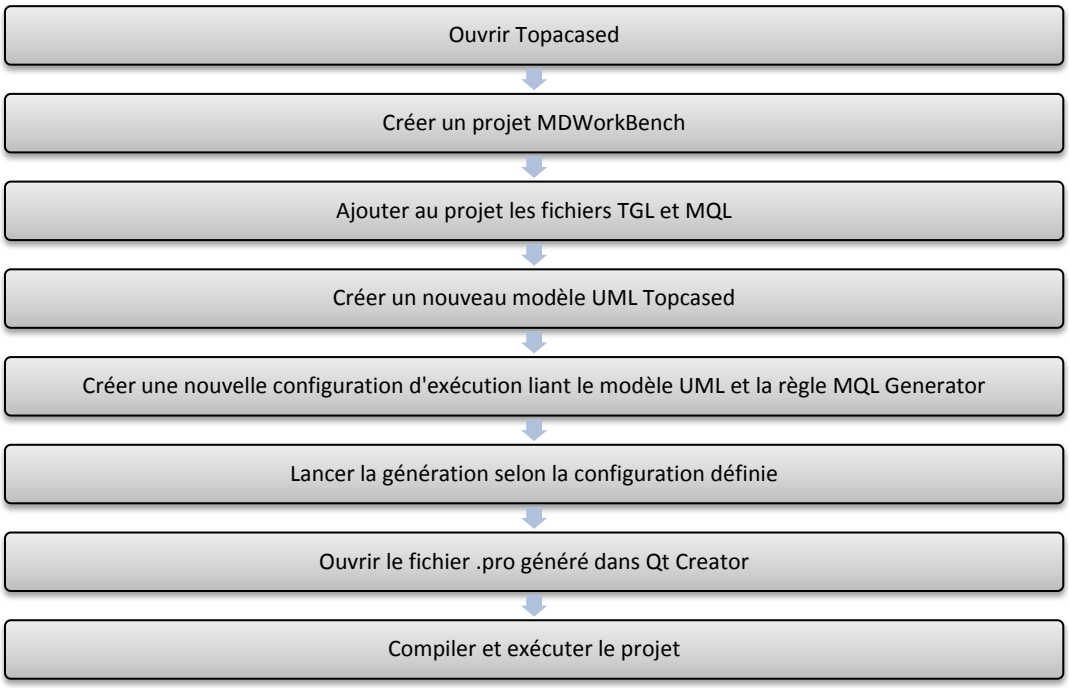


Figure 10 : Séquence des opérations

10.2. Création d'un modèle UML

10.2.1. Eléments de la norme UML utilisés

La génératrice de code a été développée dans le respect de la norme UML. Cependant seuls certains éléments relatifs à la réalisation de machines d'états-transitions sont pris en compte par le processus de génération. En conséquence, il est important lors de la création d'un modèle de n'utiliser que ces éléments :

- <Model>
- <Class>
- <State Machine>
- <Region>

- <Pseudostate>
- <State>
- <Opaque Behavior>
- <Final State>
- <Transition>
- <Trigger>
- <Instance Specification>
- <Association>
- <Property>
- <Literal Unlimited Natural>
- <Literal Integer>
- <Slot>
- <Instance Value>
- <Signal Event>
- <Signal>
- <Time Event>

10.2.2. Diagramme de classes

La première étape pour la création d'un modèle est de céer un diagramme de classe. Celui-ci peut être réalisé facilement au moyen de l'éditeur graphique de Topcased.

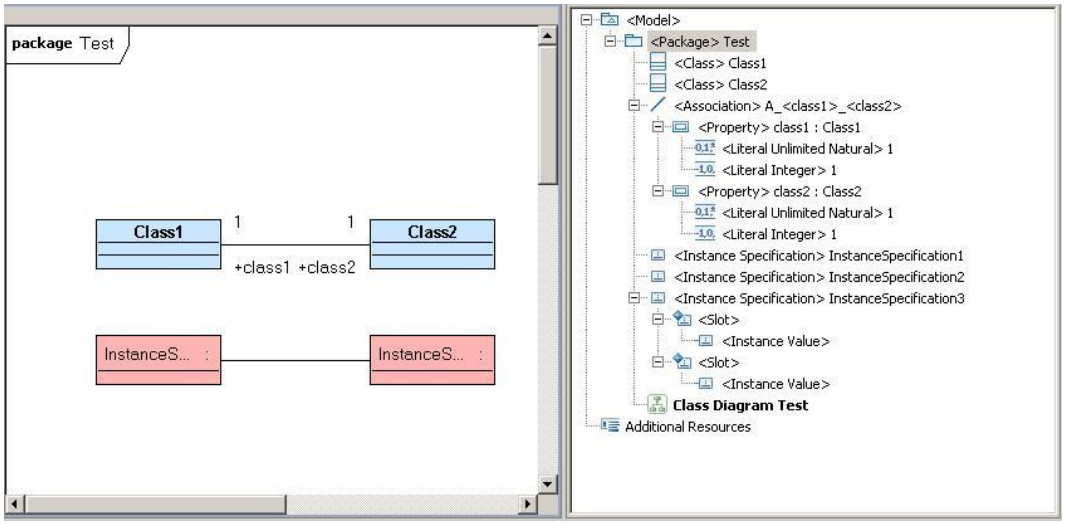


Figure 11 : Diagramme de classes

Doivent figurer sur ce diagramme :

- Toutes les classes qui contiennent une machine d'états-transitions

- Les associations entre ces classes
- Une ou plusieurs instances de chacune de ces classes
- Les liens qui instancient les associations précitées

Une fois ces éléments dessinés, l'éditeur graphique et l'éditeur en arborescence de Topcased ressemblent à la capture d'écran de la Figure 11.

Les instances doivent être associées à leurs classes, avant que les liens ne soient dessinés, sous peine que ceux-ci ne soient pas créés correctement. Ceci peut être réalisé en cliquant sur l'instance, puis en réglant la propriété « isInstanceOf » de l'onglet « Propriétés ».

Il faut également définir la navigabilité des associations. Pour ce faire, il faut sélectionner les classes appropriées dans l'attribut « Navigable Owned End » dans l'onglet « Propriétés » de chaque association.

Il est également possible de décider si une classe doit bénéficier d'un processus dédié où non en cliquant sur al classe, puis en réglant la propriété « isActive » de la section « Advanced » de l'onglet « Propriétés.

10.2.3. Diagrammes de machines d'états-transitions

Une option du menu contextuel de l'éditeur en arborescence permet d'ajouter à chaque classe un diagramme de machine d'états-transitions. Il est bon de systématiquement nommer les éléments ajoutés par l'intermédiaire de l'éditeur en arborescence.

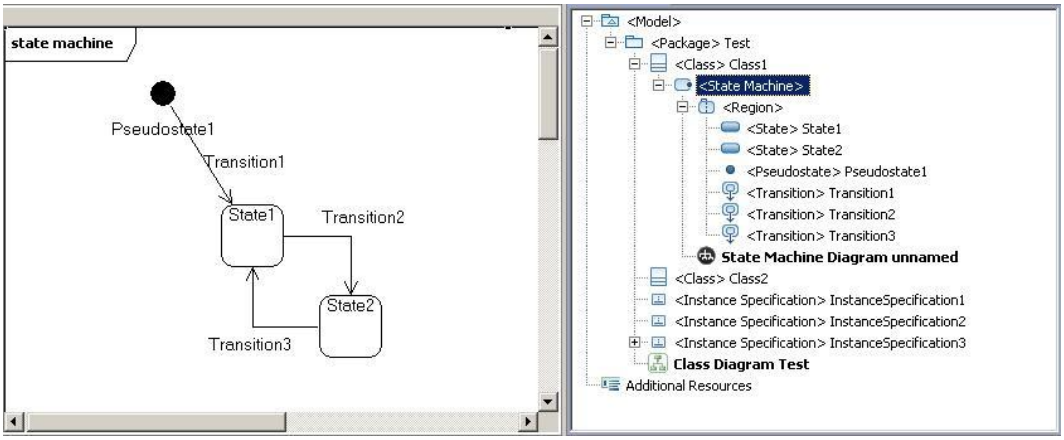


Figure 12 : Diagramme de machines d'états-transitions

L'éditeur graphique permet de dessiner facilement une machine d'etats-transitions. Peuvent figurer dans ce diagramme :

- Etats, éventuellement imbriqués
- Transitions entre ces états
- Pseudo-états initiaux et finaux
- Régions concurrentes

Une fois ces éléments dessinés, l'éditeur graphique et l'éditeur en arborescence de Topcased ressemblent à la capture d'écran de la Figure 12.

10.2.4. Ajout des triggers

Le menu contextuel de l'éditeur en arborescence permet de rajouter au modèle des <Signal Event> et <Time Event>. Ces éléments servent de déclencheurs pour les transitions.

Les <Signal Event> doivent référencer un <Signal> dans leur attribut « Signal », pour être valides. Il faut donc ajouter un <Signal> au modèle par le biais du menu contextuel, puis l'assigner au <Signal Event> dans l'onglet « Propriétés ».

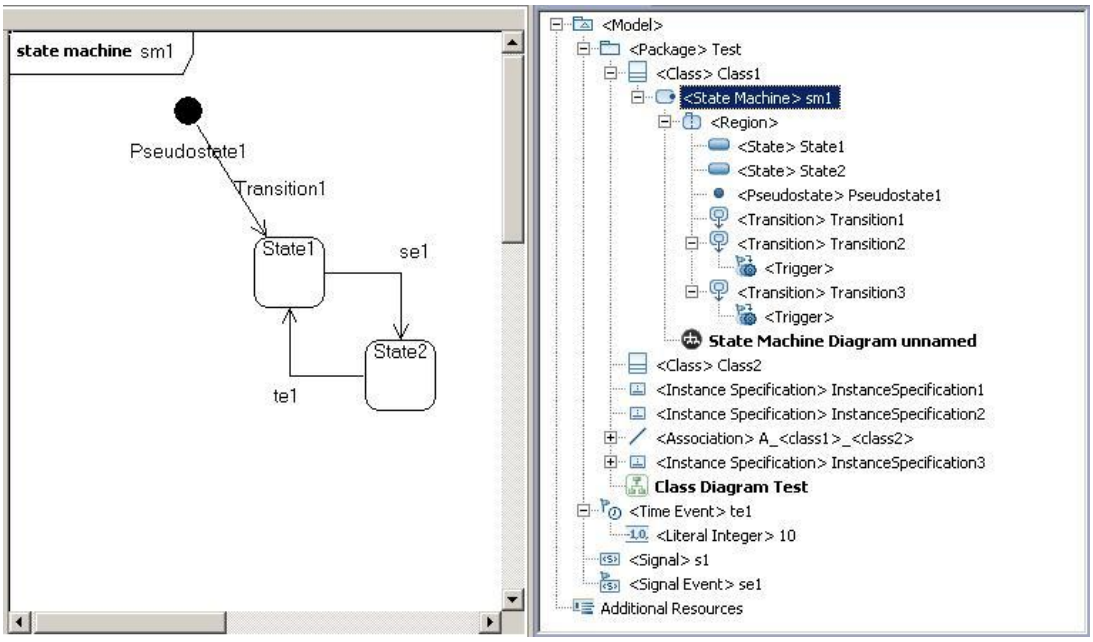


Figure 13 : Ajout des triggers

Les <Time Event> doivent être propriétaires d'un <Literal Integer> dans lequel la durée du timer peut être spécifiée par le biais de l'attribut « Value ». Il faut donc ajouter un <Literal Integer> au <Time Event>par le biais du menu contextuel, puis définir sa valeur dans l'onglet « Propriétés ».

Les événements ainsi créés doivent ensuite être assignés aux transitions. Pour ce faire, l'éditeur en arborescence permet de rajouter un <Trigger> à chaque transition. Par le biais de l'onglet « Propriétés », chaque <Trigger> reçoit dans sont attribut « Event » une référence à l'un des événements préalablement créés.

Une fois ces opérations effectuées, l'éditeur graphique et l'éditeur en arborescence de Topcased ressemblent à la capture d'écran de la Figure 13.

10.2.5. Ajout des guards, et des actions d'entrée, sortie, transition

Les guards, actions d'entrée et de sortie sont des expressions que l'utilisateur entre en dur dans le modèle. Ils sont contenus dans des <Opaque Behavior> pour les actions, et dans des <Opaque Expression> pour les guards.

Pour les actions, il faut donc ajouter un <Opaque Behavior> aux états ou aux transitions par le biais du menu contextuel de l'éditeur en arborescence, puis entrer le code voulu dans son attribut « Body » via l'onglet « Propriétés ».

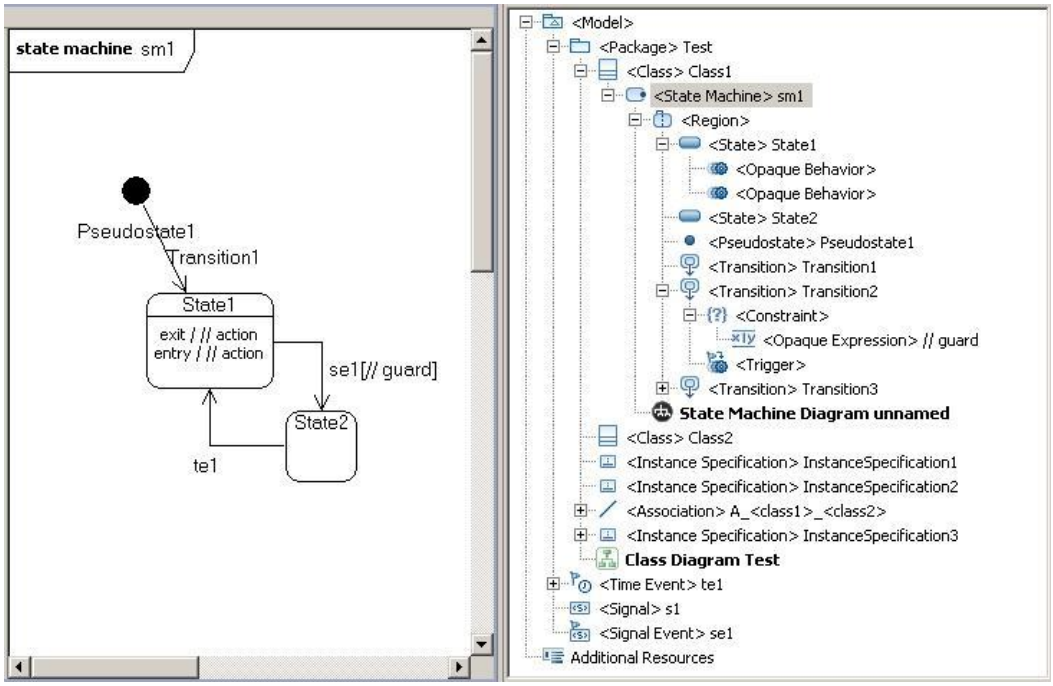


Figure 14 : Ajout des guards et actions

Le procédé est similaire pour les guards, si ce n'est que la transition concernée doit contenir un <Constraint>, dans lequel il est possible de placer la <Opaque Expression>.

Une fois ces opérations effectuées, l'éditeur graphique et l'éditeur en arborescence de Topcased ressemblent à la capture d'écran de la Figure 14.

10.2.6. Validation du modèle

Avant de lancer la génération, il est conseillé de lancer l'outil de validation de modèle intégré à Topcased. La validation ne constitue pas une garantie absolue que la génération va fonctionner, mais il est en revanche quasiment certain qu'un modèle non-valide ne sera pas généré correctement.

10.3. Tests

Les temps à disposition pour les tests s'est révélé extrèmement bref. La génératrice n'a été soumise qu'à deux modèles. Dans les deux cas la génération a pu être effectuée.

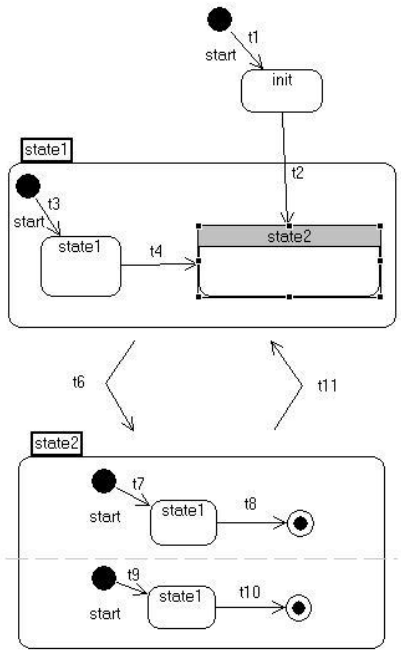


Figure 15 : Première machine de test

Le code généré à partir du premier modèle (DefaultName.uml), dont de diagramme de machine d'états-transitions est visible sur la Figure 15, se comporte correctement en exécution. Il valide donc le projet pour des machines d'états complexes, mais dont les transitions sont toutes par défaut.

Par contre, la seconde machine d'état (Minimal.uml), dont les diagrammes UML sont visibles sur la Figure 16, n'a pas pu s'exécuter correctement. L'exécution s'interrompt

systématiquement lors de l'utilisation d'un timer, ce qui amène à conclure que leur gestion n'est pas fonctionnelle. La gestion des événements est également fortement soupçonnée de fonctionner de manière déficiente, mais le temps à disposition ne permet pas de s'en assurer.

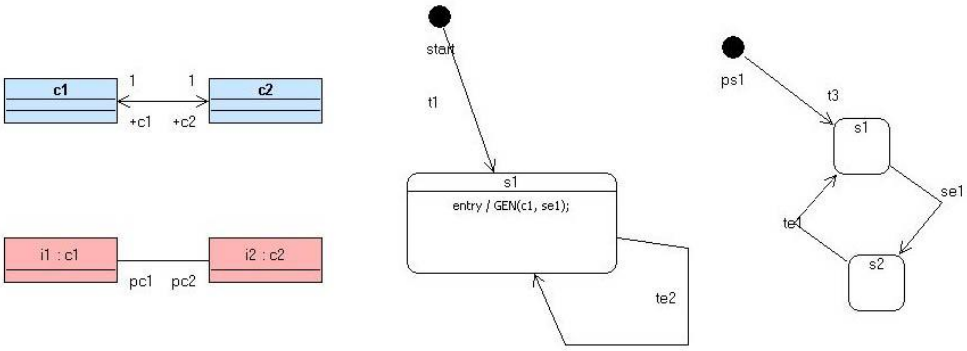


Figure 16 : Deuxième modèle de test

10.4. Erreurs connues

La gestion des transitions temporisées ne fonctionne pas. Son principe semble valide, l'erreur réside donc certainement dans la génération de code.

Les tests laissent penser que la gestion des événements ne fonctionne que dans certains cas particuliers. Une plus grande quantité de tests serait toutefois nécessaire pour en acquérir la certitude.

L'instanciation des liens entre les machines d'états-transitions est basée sur le nom des classes. Cette méthode est sémantiquement erronnée. Cette instanciation devrait être basée sur le nom des instances plutôt que sur le nom des classes elles-mêmes.

11. Conclusion

Le QXF réalisé en première partie de ce travail s'est malheureusement révélé inutile au cours du développement du projet. Ce travail a cependant permis à l'auteur de ces lignes de parfaire sa compréhension de la structure d'un XF, ainsi que d'étudier les diverses options d'implémentation des machines d'états-transitions. Ces connaissances se sont révélées très utiles pour la suite du travail.

La bonne compréhension des parties de la spécification UML concernées par le projet ont demandé un temps important. Cette étude était cependant nécessaire, afin d'implémenter les machines d'états-transitions et de générer le code correspondant dans le respect de la norme.

L'apprentissage des langages TGL et MQL s'est révélé ardu. Les puissantes possibilités de ces outils ne s'étant révélées qu'au cours du développement, il est clair que leur utilisation dans le cadre de ce projet n'est pas optimale, bien que fonctionnelle.

Au terme du temps imparti au développement de ce projet, une version de travail de la génératrice de code est fonctionnelle, malgré les obstacles précités. Celle-ci permet de créer le code nécessaire à l'exécution de machines d'états-transitions décrites par un fichier XMI, en exploitant les classes et mécanismes offerts par Qt. Les machines d'états-transitions générées ne doivent cependant pas contenir de transitions déclenchées par timer, car celles-ci induisent un bug qui n'a pu être corrigé avant le terme du projet.

12. Développements futurs

La correction du bug relatif aux transitions déclenchées par timer est un objectif prioritaire d'un éventuel développement ultérieur du projet. De même, les erreurs d'interprétation du modèle mentionées au § 10.4 doivent impérativement être rectifiées.

La qualité des requêtes MQL ainsi que la structure des templates TGL sont perfectibles. Les « transient links » n'ont par exemple pas été exploités, alors que leur utilisation est susceptible de simplifier considérablement le code. On notera également que ces fichiers sont dépourvus de commentaires. Cet état de fait est peu problématique, étant donné que les balises TGL sont très simples, et que les requêtes MQL sont facilement compréhensibles au premier coup d'oeil. Certains éléments de code, notamment parmi les scripts MQL, mériteraient toutefois quelques commentaires.

Le temps à disposition n'a permis qu'un nombre de tests insuffisant pour valider la robustesse de la génératrice. Si le développement du projet est poursuivi, il est indispensable de procéder à des tests plus systématiques.

L'utilisation de la chaine d'outils développée est rébarbative pour l'utilisateur. Les divers éléments sont mal intégrés et nécessitent beaucoup d'opérations manuelles évitables. Il serait avisé de créer une interface unifiée et simple permettant à l'utilisateur de travailler avec un maximum d'efficacité.

Dans le cas où le modèle est validé par Topcased, mais contient des cas de figure non traités par la génératrice, rien n'est prévu pour en informer l'utilisateur. Ceci mène à des erreurs peu explicites pour l'utilisateur, soit dans la génération du code, soit dans sa compilation. Il est important de remédier à ce problème, afin d'améliorer l'efficacité de la correction des erreurs de modélisation.

13. Remerciements

Je tiens à remercier toutes les personnes ci-dessous pour l'aide précieuse qu'ils m'ont apporté dans la réalisation de ce projet :

- M. Medard Rieder — Professeur à la HES-SO // Valais
- M. Thomas Sterren — Assistant scientifique à la HES-SO // Valais
- M. Nils Chatton — Assistant scientifique à la HES-SO // Valais
- M. Rico Steiner — Assistant scientifique à la HES-SO // Valais
- M. Michael Clausen — Assistant scientifique à la HES-SO // Valais

14. Bibliographie

Nokia Corporation. (2010). *Object Trees and Object Ownership*. Consulté le juillet 8, 2010, sur http://doc.qt.nokia.com/4.6/objecttrees.html

Nokia Corporation. (2010). *Signals and Slots.* Consulté le juillet 8, 2010, sur http://doc.qt.nokia.com/4.6/signalsandslots.html

Nokia Corporation. (2010). *The State Machine Framework*. Consulté le juillet 6, 2010, sur http://doc.qt.nokia.com/4.6/statemachine-api.html

Nunes, H. (2008). *Automatic Designer.* Sion: HES-SO//Valais.

Rieder, M., & Steiner, R. (2009, décembre 15). Execution Framework. Sion.

Rieder, M., & Steiner, R. (2009, septembre 15). Machines d'états-transitions et l'implémentation des machines d'états-transitions. Sion: HES-SO // Valais.

ANNEXES

1. Documentation du QXF

QxfDefaultEvent Class Reference

Event triggering default transitions.

`#include <qxfdefaultevent.h>`

Public Member Functions

QxfDefaultEvent (QxfReactive *pTarget)

Detailed Description

Event triggering default transitions.

Author:
Jean Iwanowski

Version:
1.0

This class defines a particular event that is sent whenever there is a default transition.
Definition at line 14 of file qxfdefaultevent.h.

Constructor & Destructor Documentation

QxfDefaultEvent::QxfDefaultEvent (QxfReactive * *pTarget*)

Definition at line 3 of file qxfdefaultevent.cpp.

The documentation for this class was generated from the following files:

D:/My Documents/Travail de diplôme/QXF/Qxf/qxfdefaultevent.h
D:/My Documents/Travail de diplôme/QXF/Qxf/qxfdefaultevent.cpp

QxfEvent Class Reference

Abstraction of an event.

```
#include <qxfevent.h>
```

Public Member Functions

QxfEvent (QxfReactive *pTarget)
Constructor of the QxfEvent.
QxfReactive * getTarget ()

Protected Attributes

QxfReactive * pTarget

Detailed Description

Abstraction of an event.

Author:
Jean Iwanowski

Version:
1.0

State machine specific events are derived from this class.
Definition at line 15 of file qxfevent.h.

Constructor & Destructor Documentation

QxfEvent::QxfEvent (QxfReactive * *pTarget*)

Constructor of the QxfEvent.

Parameters:
pTarget pointer on the QxfReactive to which the event must be sent
Definition at line 3 of file qxfevent.cpp.

Member Function Documentation

QxfReactive * QxfEvent::getTarget ()

Definition at line 8 of file qxfevent.cpp.

Member Data Documentation

QxfReactive* QxfEvent::pTarget `[protected]`

Definition at line 19 of file qxfevent.h.

The documentation for this class was generated from the following files:

D:/My Documents/Travail de diplôme/QXF/Qxf/qxfevent.h
D:/My Documents/Travail de diplôme/QXF/Qxf/qxfevent.cpp

QxfEventDispatcher Class Reference

Dispatcher for incoming events.

```
#include <qxfeventdispatcher.h>
```

Public Slots

void dispatchEvent ()
Sends a message to the right state machine.

Signals

void eventReceived ()
Signals that there is an event in the QxfEventQueue.

Public Member Functions

QxfEventDispatcher ()
void enqueueEvent (QxfEvent *pEvent)
Puts an event into the QxfEventQueue.

Detailed Description

Dispatcher for incoming events.

Author:
Jean Iwanowski

Version:
1.0

This class serializes incoming events for the one or many QxfReactive classes that live in a QxfThread. Therefore each QxfThread that runs QxfReactive classes must own a QxfEventDispatcher.

Definition at line 19 of file qxfeventdispatcher.h.

Constructor & Destructor Documentation

QxfEventDispatcher::QxfEventDispatcher ()

Definition at line 3 of file qxfeventdispatcher.cpp.

Member Function Documentation

void QxfEventDispatcher::dispatchEvent () [slot]

Sends a message to the right state machine.
Definition at line 20 of file qxfeventdispatcher.cpp.

void QxfEventDispatcher::enqueueEvent (QxfEvent * *pEvent*)

Puts an event into the QxfEventQueue.

Parameters:
pEvent Pointer on the QxfEvent to enqueue
Definition at line 13 of file qxfeventdispatcher.cpp.

void QxfEventDispatcher::eventReceived () [signal]

Signals that there is an event in the QxfEventQueue.

The documentation for this class was generated from the following files:
D:/My Documents/Travail de diplôme/QXF/Qxf/qxfeventdispatcher.h
D:/My Documents/Travail de diplôme/QXF/Qxf/qxfeventdispatcher.cpp

QxfFactory Class Reference

Factory of the QXF.

```
#include <qxffactory.h>
```

Public Member Functions

QxfFactory ()
~QxfFactory ()

Detailed Description

Factory of the QXF.

Author:
Jean Iwanowski

Version:
1.0

This class is responsible for the instanciation of all necessary classes for the QXF to run properly. It is run by Qt's main thread. It creates a QxfThread for the time management, and as many QxfThread as necessary to run all the active state machines of a specific model. The QxfTimeoutManager, all the QxfEventDispatcher and QxfReactive are also instantiated in this Factory so they can be linked together. Then they are pushed into their respective threads, for pseudo-parallel execution.

Definition at line 25 of file qxffactory.h.

Constructor & Destructor Documentation

QxfFactory::QxfFactory ()

Definition at line 3 of file qxffactory.cpp.

QxfFactory::~QxfFactory ()

Definition at line 42 of file qxffactory.cpp.

The documentation for this class was generated from the following files:

D:/My Documents/Travail de diplôme/QXF/Qxf/qxffactory.h
D:/My Documents/Travail de diplôme/QXF/Qxf/qxffactory.cpp

QxfReactive Class Reference

Abstraction of a state machine.

```
#include <qxfreactive.h>
```

Signals

void sendEvent (QxfEvent *)
Sends an event.

void sendTimeout (QxfTimeoutEvent *)
Sends a timer to the QxfTimeoutManager.
void stateChanged (QxfReactive *)
Signals that a transition occured.

Public Member Functions

QxfReactive (QxfEventDispatcher *pEventDispatcher, QxfTimeoutManager *pTimeoutManager)
constructor of the QxfReactive
~QxfReactive ()
virtual void behavior (QxfEvent *pEvent)=0
Defines the behavior of the state machine.
virtual void initRelations (QxfReactive *pReactive)=0
Links states machines together.
void startBehavior ()
Starts state machine.
virtual int typeOf (QxfEvent *pEvent)=0

Protected Member Functions

template<typename T > void createEvent (QxfReactive *pTarget)
Creates an event.
void createTimeout (int value)
Creates an event.

Detailed Description

Abstraction of a state machine.

Author:
Jean Iwanowski

Version:
1.0

State machines which are specific to a given model must derive from this class.
Definition at line 23 of file qxfreactive.h.

Constructor & Destructor Documentation

QxfReactive::QxfReactive (QxfEventDispatcher * *pEventDispatcher*, QxfTimeoutManager * *pTimeoutManager*)

constructor of the QxfReactive

Parameters:
pEventDispatcher Pointer on the QxfEventDispatcher which manages events received by the state machine
pTimeoutManager Pointer on the QxfTimeoutManager

Definition at line 3 of file qxfreactive.cpp.

QxfReactive::~QxfReactive ()

Definition at line 30 of file qxfreactive.cpp.

Member Function Documentation

virtual void QxfReactive::behavior (QxfEvent * *pEvent*) `[pure virtual]`

Defines the behavior of the state machine.

This function reacts to an incoming QxfEvent, according to the behavior defined in the model.

template<typename T > void QxfReactive::createEvent (QxfReactive * *pTarget*) `[inline, protected]`

Creates an event.

Parameters:

pTarget Pointer on the QxfReactive to which the event must be sent

Template Parameters:

T Type of the event to create

This function creates an event of specified type, and sends it to its destination through signal-slot mechanism.

Definition at line 41 of file qxfreactive.h.

void QxfReactive::createTimeout (int *value*) `[protected]`

Creates an event.

Parameters:

value Number of ticks until timeout

This function creates a timeout event of specified type, and sends it to the QxfTimeoutManager through signal-slot mechanism.

Definition at line 34 of file qxfreactive.cpp.

virtual void QxfReactive::initRelations (QxfReactive * *pReactive*) `[pure virtual]`

Links states machines together.

Parameters:

pReactive Pointer on the QxfReactive object to link to the state machine

This function binds state machines together according to the model.

void QxfReactive::sendEvent (QxfEvent *) [signal]

Sends an event.

Parameters:

_t1 Pointer on the QxfEvent to be sent

void QxfReactive::sendTimeout (QxfTimeoutEvent *) [signal]

Sends a timer to the QxfTimeoutManager.

Parameters:

_t1 Pointer on the QxfTimeoutEvent to be sent

void QxfReactive::startBehavior ()

Starts state machine.

This function creates a QxfStartBehaviorEvent and sends it to the state machine.

Definition at line 45 of file qxfreactive.cpp.

void QxfReactive::stateChanged (QxfReactive *) [signal]

Signals that a transition occured.

Parameters:

_t1 Pointer on the QxfReactive object whose state changed

virtual int QxfReactive::typeOf (QxfEvent * *pEvent*) [pure virtual]

Parameters:

Finds the type of a QxfEvent object
pEvent Pointer on the QxfEvent to analyse

Returns:

Type of the event, coded as an integer

This function compares a QxfEvent with all possible event types through pointer casting until there is a match.

The documentation for this class was generated from the following files:

D:/My Documents/Travail de diplôme/QXF/Qxf/qxfreactive.h
D:/My Documents/Travail de diplôme/QXF/Qxf/qxfreactive.cpp

QxfStartBehaviorEvent Class Reference

Event capable of starting a state machine.

`#include <qxfstartbehaviorevent.h>`

Public Member Functions

QxfStartBehaviorEvent (QxfReactive *pTarget)
Constructor of the QxfStartBehaviorEvent.

Detailed Description

Event capable of starting a state machine.

Author:
Jean Iwanowski

Version:
1.0

This class defines a particular event that must be sent to any QxfReactive to start its behavior.
Definition at line 14 of file qxfstartbehaviorevent.h.

Constructor & Destructor Documentation

QxfStartBehaviorEvent::QxfStartBehaviorEvent (QxfReactive * *pTarget*)

Constructor of the QxfStartBehaviorEvent.

Parameters:
pTarget pointer on the QxfReactive to which the event must be sent
Definition at line 3 of file qxfstartbehaviorevent.cpp.

The documentation for this class was generated from the following files:
D:/My Documents/Travail de diplôme/QXF/Qxf/qxfstartbehaviorevent.h
D:/My Documents/Travail de diplôme/QXF/Qxf/qxfstartbehaviorevent.cpp

QxfThread Class Reference

Thread management interface for the Qxf.

`#include <qxfthread.h>`

Public Member Functions

QxfThread (Priority p)
Constructor of the QxfThread.
~QxfThread ()

Detailed Description

Thread management interface for the Qxf.

Author:
Jean Iwanowski

Version:
1.0

Provides an active loop in which classes can live independently from other QXF components.
Definition at line 14 of file qxfthread.h.

Constructor & Destructor Documentation

QxfThread::QxfThread (Priority *p*)

Constructor of the QxfThread.

Parameters:
p Priority level of the thread, of type QThread::Priority
Definition at line 3 of file qxfthread.cpp.

QxfThread::~QxfThread ()

Definition at line 8 of file qxfthread.cpp.

The documentation for this class was generated from the following files:
D:/My Documents/Travail de diplôme/QXF/Qxf/qxfthread.h

D:/My Documents/Travail de diplôme/QXF/Qxf/qxfthread.cpp

QxfTimeoutEvent Class Reference

Event signaling that a timeout is reached.

`#include <qxftimeoutevent.h>`

Public Member Functions

QxfTimeoutEvent (int value, QxfReactive *pSource)
Constructor of the QxfTimeoutEvent.
int getTimeoutTicks ()
int getRemainingTicks ()
void setRemainingTicks (int ticks)
void decrement ()
Decrements remaining ticks counter.

Detailed Description

Event signaling that a timeout is reached.

Author:
Jean Iwanowski

Version:
1.0

This class defines a particular event that is sent to any QxfReactive object that uses a timed transition once the timeout is reached.

Definition at line 16 of file qxftimeoutevent.h.

Constructor & Destructor Documentation

QxfTimeoutEvent::QxfTimeoutEvent (int *value*, QxfReactive * *pSource*)

Constructor of the QxfTimeoutEvent.

Parameters:
value Number of ticks until timeout
pSource Pointer on the QxfReactive object that started the timeout

Definition at line 3 of file qxftimeoutevent.cpp.

Member Function Documentation

void QxfTimeoutEvent::decrement ()

Decrements remaining ticks counter.
Definition at line 25 of file qxftimeoutevent.cpp.

int QxfTimeoutEvent::getRemainingTicks ()

Definition at line 15 of file qxftimeoutevent.cpp.

int QxfTimeoutEvent::getTimeoutTicks ()

Definition at line 10 of file qxftimeoutevent.cpp.

void QxfTimeoutEvent::setRemainingTicks (int *ticks*)

Definition at line 20 of file qxftimeoutevent.cpp.

The documentation for this class was generated from the following files:

D:/My Documents/Travail de diplôme/QXF/Qxf/qxftimeoutevent.h
D:/My Documents/Travail de diplôme/QXF/Qxf/qxftimeoutevent.cpp

QxfTimeoutList Class Reference

List of pending QxfTimeoutEvent.

```
#include <qxftimeoutlist.h>
```

Public Member Functions

QxfTimeoutList ()
void add (QxfTimeoutEvent *pTimeoutEvent)
Adds a QxfTimeoutEvent at the right place in the list.
void remove (QxfReactive *pTarget)
Removes QxfTimeoutEvent with specific target.
void print ()

Detailed Description

List of pending QxfTimeoutEvent.

Author:
Jean Iwanowski

Version:
1.0

This class stores pending QxfTimeoutEvent by order of imminence.
Definition at line 14 of file qxftimeoutlist.h.

Constructor & Destructor Documentation

QxfTimeoutList::QxfTimeoutList ()

Definition at line 4 of file qxftimeoutlist.cpp.

Member Function Documentation

void QxfTimeoutList::add (QxfTimeoutEvent * *pTimeoutEvent*)

Adds a QxfTimeoutEvent at the right place in the list.

Parameters:
pTimeoutEvent Pointer on the QxfTimeoutEvent to add.
Definition at line 17 of file qxftimeoutlist.cpp.

void QxfTimeoutList::print ()

Definition at line 8 of file qxftimeoutlist.cpp.

void QxfTimeoutList::remove (QxfReactive * *pTarget*)

Removes QxfTimeoutEvent with specific target.

Parameters:
pTarget Pointer on the target of the QxfTimeoutEvent to remove
Definition at line 66 of file qxftimeoutlist.cpp.

The documentation for this class was generated from the following files:
D:/My Documents/Travail de diplôme/QXF/Qxf/qxftimeoutlist.h
D:/My Documents/Travail de diplôme/QXF/Qxf/qxftimeoutlist.cpp

QxfTimeoutManager Class Reference

Timeout manager of the QXF.

`#include <qxftimeoutmanager.h>`

Public Slots

void tick ()
Decrements all registered timers.

void receiveTimeout (QxfTimeoutEvent *pTimeoutEvent)
Adds a timer to the QxfTimeoutList.

void clearTimeouts (QxfReactive *pTarget)
Clears useless timeouts.

Signals

void sendEvent (QxfEvent *)
Sends an event.

Public Member Functions

QxfTimeoutManager (int ms)
Constructor of the QxfTimeoutManager.

Detailed Description

Timeout manager of the QXF.

Author:
Jean Iwanowski

Version:
1.0

Starts a timer whenever requested by any class, and sends a timeout event to the target QxfReactive class when the timeout is reached.

Definition at line 18 of file qxftimeoutmanager.h.

Constructor & Destructor Documentation

QxfTimeoutManager::QxfTimeoutManager (int *ms*)

Constructor of the QxfTimeoutManager.

Parameters:

ms Number of miliseconds per tick

Definition at line 4 of file qxftimeoutmanager.cpp.

Member Function Documentation

void QxfTimeoutManager::clearTimeouts (QxfReactive * *pTarget*) [slot]

Clears useless timeouts.

Parameters:

pTarget Pointer on the the target of the timeouts that must be removed.

Definition at line 40 of file qxftimeoutmanager.cpp.

void QxfTimeoutManager::receiveTimeout (QxfTimeoutEvent * *pTimeoutEvent*) [slot]

Adds a timer to the QxfTimeoutList.

Parameters:

pTimeoutEvent Pointer on the timer to add to list

Definition at line 45 of file qxftimeoutmanager.cpp.

void QxfTimeoutManager::sendEvent (QxfEvent *) [signal]

Sends an event.

Parameters:

_t1 Pointer on the QxfEvent to send

void QxfTimeoutManager::tick () [slot]

Decrements all registered timers.

Definition at line 15 of file qxftimeoutmanager.cpp.

The documentation for this class was generated from the following files:

D:/My Documents/Travail de diplôme/QXF/Qxf/qxftimeoutmanager.h

D:/My Documents/Travail de diplôme/QXF/Qxf/qxftimeoutmanager.cpp

2. Classe « QXFReactive »

Fichier d'en-têtes

```
#ifndef QXFREACTIVE_H
#define QXFREACTIVE_H

#define GEN(event, target)      ( createEvent< event >( &target ) )

#include <QObject>
#include <qxftimeoutmanager.h>
#include <qxfeventdispatcher.h>
#include <qxfstartbehaviorevent.h>

class QxfEvent;
class QxfTimeoutEvent;
class QxfEventDispatcher;

/*!
* \brief    Abstraction of a state machine
* \author   Jean Iwanowski
* \version  1.0
*
* State machines which are specific to a given model must derive from this
* class.
*/
class QxfReactive : public QObject
{
    Q_OBJECT
private:
    // pointer on the thread in which the reactive entity lives
    QxfEventDispatcher * pEventDispatcher;

protected:
    /*!
  * \brief    Creates an event
  * \param    pTarget Pointer on the QxfReactive to which the event must
  *           be sent
  * \tparam   T Type of the event to create
  *
  * This function creates an event of specified type, and sends it to its
  * destination through signal-slot mechanism.
  */
   template<typename T>
   void createEvent(QxfReactive* pTarget)
   {
      // creates a connexion with target
       connect(
               this,
               SIGNAL( sendEvent( QxfEvent* ) ),
               pTarget,
               SLOT( receiveEvent( QxfEvent* ) ),
               Qt::QueuedConnection
               );
       // creates the event to be sent
       QxfEvent* pEvent = new T (pTarget);
       // sends the event
       emit sendEvent(pEvent);
       // ends the connextion with target
       disconnect(
               this,
               SIGNAL( sendEvent( QxfEvent* ) ),
```

```
                pTarget,
                SLOT( receiveEvent( QxfEvent* ) )
                );
    }

    /*!
   *  \brief     Creates an event
   *  \param     value Number of ticks until timeout
   *
   *  This function creates a timeout event of specified type, and sends it to
   *  the QxfTimeoutManager through signal-slot mechanism.
   */
     void createTimeout( int value );

public:
     /*!
    *  \brief     constructor of the QxfReactive
       \param     pEventDispatcher Pointer on the QxfEventDispatcher which
    *          manages events received by the state machine
    *  \param     pTimeoutManager Pointer on the QxfTimeoutManager
    */
      QxfReactive( QxfEventDispatcher * pEventDispatcher,
                   QxfTimeoutManager * pTimeoutManager );

      // destructor
      ~QxfReactive();

     /*!
    *  \brief     Defines the behavior of the state machine
    *
    *  This function reacts to an incoming QxfEvent, according to the behavior
    *  defined in the model.
    */
      virtual void behavior( QxfEvent* pEvent ) = 0;

    /*!
    *  \brief     Links states machines together
    *  \param     pReactive Pointer on the QxfReactive object to link to the
    *          state machine
    *
    *  This function binds state machines together according to the model.
    */
      virtual void initRelations( QxfReactive* pReactive ) = 0;

      /*!
    *  \brief     Starts state machine.
    *
    *  This function creates a QxfStartBehaviorEvent and sends it to the state
    *  machine.
    */
      void startBehavior();

      /*!
    *  \param     Finds the type of a QxfEvent object
    *  \param     pEvent Pointer on the QxfEvent to analyse
    *  \return    Type of the event, coded as an integer
    *
    *  This function compares a QxfEvent with all possible event types through
    *  pointer casting until there is a match.
    */
      virtual int typeOf( QxfEvent* pEvent ) = 0;
```

```
signals:
    /*!
  *  \brief     Sends an event
  *  \param     _t1 Pointer on the QxfEvent to be sent
  */
    void sendEvent( QxfEvent* );

    /*!
  *  \brief     Sends a timer to the QxfTimeoutManager
  *  \param     _t1 Pointer on the QxfTimeoutEvent to be sent
  */
    void sendTimeout( QxfTimeoutEvent* );

    /*!
  *  \brief     Signals that a transition occured
  *  \param     _t1 Pointer on the QxfReactive object whose state changed
  */
    void stateChanged( QxfReactive* );

private slots :
    /*!
  *  \brief     Receives an event
  *  \param     pEvent Pointer on the incoming QxfEvent
  *
  *  Receives a QxfEvent and passes it to the QxfEventDispatcher.
  */
    void receiveEvent( QxfEvent * pEvent );
};

#endif // QXFREACTIVE_H
```

Fichier de définitions

```
#include "qxfreactive.h"

QxfReactive::QxfReactive( QxfEventDispatcher* pEventDispatcher,
                          QxfTimeoutManager* pTimeoutManager )
{
    // attributes initialisation
    this->pEventDispatcher = pEventDispatcher;

    // signals-slots connexions
    connect(
            this,
            SIGNAL( sendTimeout( QxfTimeoutEvent * ) ),
            pTimeoutManager,
            SLOT( receiveTimeout( QxfTimeoutEvent * ) )
            );
    connect(
            pTimeoutManager,
            SIGNAL( sendEvent( QxfEvent * ) ),
            this,
            SLOT( receiveEvent( QxfEvent * ) )
            );
    connect(
            this,
            SIGNAL( stateChanged( QxfReactive * ) ),
            pTimeoutManager,
            SLOT( clearTimeouts( QxfReactive * ) )
            );
}

QxfReactive::~QxfReactive()
{
}

void QxfReactive::createTimeout( int value )
{
    QxfTimeoutEvent * pTimeoutEvent = new QxfTimeoutEvent( value, this );
    emit sendTimeout( pTimeoutEvent );
}

void QxfReactive::receiveEvent( QxfEvent * pEvent )
{
    pEventDispatcher -> enqueueEvent( pEvent );
}

void QxfReactive::startBehavior()
{
    GEN(QxfStartBehaviorEvent, *this);
}
```

3. Classe “QxfEventDispatcher”

Fichier d’en-têtes

```
#ifndef QXFEVENTDISPATCHER_H
#define QXFEVENTDISPATCHER_H

#include <qxfeventqueue.h>
#include <qxfevent.h>
#include <qxfreactive.h>

class QxfReactive;

/*!
* \brief    Dispatcher for incoming events
* \author   Jean Iwanowski
* \version  1.0
*
* This class serializes incoming events for the one or many QxfReactive
* classes that live in a QxfThread. Therefore each QxfThread that runs
* QxfReactive classes must own a QxfEventDispatcher.
*/
class QxfEventDispatcher : public QObject
{
    Q_OBJECT
private:
    // events storage for the active class
    QxfEventQueue theEventQueue;

public:
    //constructors and destructors
    QxfEventDispatcher();

    /*!
  * \brief    Puts an event into the QxfEventQueue
  * \param    pEvent Pointer on the QxfEvent to enqueue
  */
    void enqueueEvent(QxfEvent* pEvent);

signals:
    /*!
  * \brief    Signals that there is an event in the QxfEventQueue
  */
    void eventReceived();


public slots:
    /*!
  * \brief    Sends a message to the right state machine
  */
    void dispatchEvent();
};

#endif // QXFEVENTDISPATCHER_H
```

Fichier de definitions

```
#include "qxfeventdispatcher.h"

QxfEventDispatcher::QxfEventDispatcher()
{
    connect(
            this,
            SIGNAL( eventReceived() ),
            this,
            SLOT( dispatchEvent() )
            );
}

void QxfEventDispatcher::enqueueEvent(QxfEvent* pEvent)
{
    theEventQueue.enqueue(pEvent);
    emit eventReceived();
}


void QxfEventDispatcher::dispatchEvent()
{
    if(!theEventQueue.isEmpty())
    {
        QxfEvent* pEvent = theEventQueue.dequeue();
        QxfReactive* pTarget = pEvent->getTarget();
        pTarget->behavior(pEvent);
    }
}
```

4. Classe “QxfEventQueue”

Fichier d’en-têtes

```
#ifndef QXFEVENTQUEUE_H
#define QXFEVENTQUEUE_H

#include <QQueue>
#include <qxfevent.h>

typedef QQueue < QxfEvent * > QxfEventQueue;

/*
class QxfEventQueue : public QQueue < QxfEvent * >
{
public:
   QxfEventQueue();
};
*/

#endif // QXFEVENTQUEUE_H
```

5. Classe “QxfEvent”

Fichier d’en-têtes

```
#ifndef QXFEVENT H
#define QXFEVENT_H

#include <QObject>

class QxfReactive;

/*!
*  \brief     Abstraction of an event
*  \author    Jean Iwanowski
*  \version   1.0
*
*  State machine specific events are derived from this class.
*/
class QxfEvent : public QObject
{
protected:
    // pointer on the state machine to which the event is destined
    QxfReactive* pTarget;

public:
    /*!
  *  \brief     Constructor of the QxfEvent
  *  \param     pTarget pointer on the QxfReactive to which the event must
  *         be sent
  */
    QxfEvent( QxfReactive* pTarget );

    // getters and setters
    QxfReactive* getTarget();
};

#endif // QXFEVENT_H
```

Fichier de definitions

```
#include "qxfevent.h"

QxfEvent::QxfEvent( QxfReactive* pTarget)
{
    this->pTarget = pTarget;
}

QxfReactive* QxfEvent::getTarget()
{
    return pTarget;
}
```

6. Classe “QxfStartBehaviorEvent”

Fichier d’en-têtes

```
#ifndef QXFSTARTBEHAVIOREVENT_H
#define QXFSTARTBEHAVIOREVENT_H

#include <qxfevent.h>

/*!
*  \brief     Event capable of starting a state machine
*  \author    Jean Iwanowski
*  \version   1.0
*
*  This class defines a particular event that must be sent to any QxfReactive
*  to start its behavior.
*/
class QxfStartBehaviorEvent : public QxfEvent
{
public:
    /*!
  *  \brief     Constructor of the QxfStartBehaviorEvent
  *  \param     pTarget pointer on the QxfReactive to which the event must
  *         be sent
  */
    QxfStartBehaviorEvent( QxfReactive* pTarget );
};

#endif // QXFSTARTBEHAVIOREVENT_H
```

Fichier de definitions

```
#include "qxfstartbehaviorevent.h"

QxfStartBehaviorEvent::QxfStartBehaviorEvent( QxfReactive* pTarget )
        :QxfEvent( pTarget )
{
}
```

7. Classe “QxfDefaultEvent”

Fichier d’en-têtes

```
#ifndef QxfDefaultEvent_H
#define QxfDefaultEvent_H

#include <qxfevent.h>

/*!
*  \brief     Event triggering default transitions
*  \author    Jean Iwanowski
*  \version   1.0
*
*  This class defines a particular event that is sent whenever there is a
*  default transition.
*/
class QxfDefaultEvent : public QxfEvent
{
public:
    QxfDefaultEvent( QxfReactive* pTarget );
};

#endif // QXFSUBMACHINEEVENT_H
```

Fichier de definitions

```
#include "qxfdefaultevent.h"

QxfDefaultEvent::QxfDefaultEvent( QxfReactive* pTarget ):QxfEvent(pTarget)
{
}
```

8. Classe “QxfTimeoutEvent”

Fichier d’en-têtes

```
#ifndef QXFTIMEOUTEVENT_H
#define QXFTIMEOUTEVENT_H

#include <qxfevent.h>

class QxfReactive;

/*!
* \brief    Event signaling that a timeout is reached
* \author   Jean Iwanowski
* \version  1.0
*
* This class defines a particular event that is sent to any QxfReactive object
* that uses a timed transition once the timeout is reached.
*/
class QxfTimeoutEvent : public QxfEvent
{
private:
    // timer duration
    int timeoutTicks;

    // remaining ticks until timeout
    int remainingTicks;

public:
    /*!
  * \brief    Constructor of the QxfTimeoutEvent
  * \param    value Number of ticks until timeout
  * \param    pSource Pointer on the QxfReactive object that started the
  *           timeout
  */
     QxfTimeoutEvent(int value, QxfReactive* pSource);

    // getters and setters
    int getTimeoutTicks();
    int getRemainingTicks();
    void setRemainingTicks(int ticks);

    /*!
  * \brief    Decrements remaining ticks counter
  */
    void decrement();
};

#endif // QXFTIMEOUTEVENT_H
```

Fichier de definitions

```
#include "qxftimeoutevent.h"

QxfTimeoutEvent::QxfTimeoutEvent(int value, QxfReactive* pSource)
        :QxfEvent(pSource)
{
    timeoutTicks = value;
    remainingTicks = value;
}

int QxfTimeoutEvent::getTimeoutTicks()
{
    return timeoutTicks;
}

int QxfTimeoutEvent::getRemainingTicks()
{
    return remainingTicks;
}

void QxfTimeoutEvent::setRemainingTicks(int ticks)
{
    remainingTicks = ticks;
}

void QxfTimeoutEvent::decrement()
{
    if(remainingTicks > 0)
    {
        --remainingTicks;
    }
}
```

9. Classe “QxfTimeoutManager”

Fichier d’en-têtes

```
#ifndef QXFTIMEOUTMANAGER_H
#define QXFTIMEOUTMANAGER_H

#include <QObject>
#include <QTimer>
#include <qxftimeoutlist.h>
#include <qxfevent.h>
#include <qxftimeoutevent.h>

/*!
*  \brief     Timeout manager of the QXF.
*  \author    Jean Iwanowski
*  \version   1.0
*
*  Starts a timer whenever requested by any class, and sends a timeout event to
*  the target QxfReactive class when the timeout is reached.
*/
class QxfTimeoutManager : public QObject
{
    Q_OBJECT
private:
    // storage for the currently running timers
    QxfTimeoutList theTimeoutList;
    QTimer theTimer;

public:
    /*!
  *  \brief     Constructor of the QxfTimeoutManager
  *  \param     ms Number of miliseconds per tick
  */
    QxfTimeoutManager( int ms );

signals:
    /*!
  *  \brief     Sends an event
  *  \param     _t1 Pointer on the QxfEvent to send
  */
    void sendEvent( QxfEvent* );


public slots:
    /*!
  *  \brief     Decrements all registered timers
  */
    void tick();

    /*!
  *  \brief     Adds a timer to the QxfTimeoutList
  *  \param     pTimeoutEvent Pointer on the timer to add to list
  */
    void receiveTimeout( QxfTimeoutEvent* pTimeoutEvent );

    /*!
  *  \brief     Clears useless timeouts.
  *  \param     pTarget Pointer on the the target of the timeouts that must be
  *             removed.
   */
```

```
    void clearTimeouts(QxfReactive* pTarget);

};

#endif // QXFTIMEOUTMANAGER_H
```

Fichier de definitions

```
#include <QtDebug>
#include "qxftimeoutmanager.h"

QxfTimeoutManager::QxfTimeoutManager(int ms)
{
    connect(
            &theTimer,
            SIGNAL ( timeout() ),
            this,
            SLOT ( tick() )
            );
    theTimer.start( ms );
}

 void QxfTimeoutManager::tick()
{
    // checks if there is a timer to decrement
    if( !theTimeoutList.isEmpty() )
    {
        // decrements timers
        theTimeoutList.first()->decrement();
        //checks if a timeout is reached
        while(  !theTimeoutList.isEmpty()
            && theTimeoutList.first()->getRemainingTicks() == 0 )
        {
            // sends timeout event
            sendEvent( theTimeoutList.first() );
            // deletes timeout reference
            theTimeoutList.pop_front();

            // DEBUG : writes timeouts in application output
            qDebug() << "----------------------";
            qDebug() << "Timeout reached";
            theTimeoutList.print();
            qDebug() << "----------------------\n";
        }
    }
}

void QxfTimeoutManager::clearTimeouts(QxfReactive* pTarget)
{
    theTimeoutList.remove( pTarget );
}

void QxfTimeoutManager::receiveTimeout(QxfTimeoutEvent* pTimeoutEvent)
{
    theTimeoutList.add( pTimeoutEvent );
}
```

10. Classe “QxfTimeoutList”

Fichier d’en-têtes

```
#ifndef QXFTIMEOUTLIST_H
#define QXFTIMEOUTLIST_H

#include <QList>
#include <qxftimeoutevent.h>

/*!
*  \brief     List of pending QxfTimeoutEvent
*  \author    Jean Iwanowski
*  \version   1.0
*
*  This class stores pending QxfTimeoutEvent by order of imminence.
*/
class QxfTimeoutList : public QList<QxfTimeoutEvent*>
{
public:
    // constructors and destructors
    QxfTimeoutList();

    /*!
  *  \brief     Adds a QxfTimeoutEvent at the right place in the list.
  *  \param     pTimeoutEvent Pointer on the QxfTimeoutEvent to add.
  */
    void add(QxfTimeoutEvent* pTimeoutEvent);

    /*!
  *  \brief     Removes QxfTimeoutEvent with specific target.
  *  \param     pTarget Pointer on the target of the QxfTimeoutEvent to remove
  */
    void remove(QxfReactive* pTarget);

    // debug function
    void print();

};

#endif // QXFTIMEOUTLIST_H
```

Fichier de definitions

```
#include <QtDebug>
#include "qxftimeoutlist.h"

QxfTimeoutList::QxfTimeoutList()
{
}

void QxfTimeoutList::print()
{
    foreach (QxfTimeoutEvent* ev, *this)
    {
        qDebug() << "Timeout " << ev->getTimeoutTicks()
                 << ",  remaining = " << ev->getRemainingTicks();
    }
}

void QxfTimeoutList::add(QxfTimeoutEvent* pTimeoutEvent)
{
    int current = 0;
    int accumulator = 0;
    int correction = 0;
    // finds the right place where to insert the timer

    // DEBUG : writes timeouts in application output
    qDebug() << "----------------------";
    print();

    if( isEmpty() )
    {
        append( pTimeoutEvent );
    }
    else
    {
        while( current < size() &&
               at( current )->getRemainingTicks() < pTimeoutEvent-
>getRemainingTicks() )
        {
            accumulator += at( current )->getRemainingTicks();
            ++current;
        }
        // computes the correction for the remaining ticks field
        correction = pTimeoutEvent->getTimeoutTicks() - accumulator;

        // checks if the timer must be inserted or appended
        if( current < size() )
        {
            // inserts the timer in the middle of the list
            insert( current, pTimeoutEvent );
            // updates next timer remaining ticks to match the timers chain
            at( current + 1 )->setRemainingTicks(
                    at( current + 1 )->getRemainingTicks() - correction );
        }
        else
        {
            // append the timer at the end of the list
            append( pTimeoutEvent );
        }
        // updates the remaining ticks to match the timers chain
        pTimeoutEvent->setRemainingTicks( correction );
    }
    // DEBUG : writes timeouts in application output
    qDebug() << "Add timeout :";
    print();
    qDebug() << "----------------------\n";
}
```

```
void QxfTimeoutList::remove( QxfReactive* pTarget )
{
   // checks if timeouts list is not empty
    if ( !isEmpty() )
    {
        // browses the timeout list
        for( int i = 0; i < size(); ++i )
        {
            if( at( i )->getTarget() == pTarget )
            {
                if( i == size() - 1 )
                {
                    removeLast();
                }
                else
                {
                    at( i+1 )->setRemainingTicks(
                            at( i )->getRemainingTicks() + at( i+1 )-
>getRemainingTicks() );
                    removeAt(i);
                }
            }
        }
    }
}
```

11. Classe “QxfFactory”

Fichier d’en-têtes

```
#ifndef QXFFACTORY_H
#define QXFFACTORY_H

#include <QObject>
#include <QThread.h>
#include <qxfeventdispatcher.h>
#include <qxftimeoutmanager.h>
#include <qxfreactive.h>
#include <simplestatemachine1.h>
#include <simplestatemachine2.h>

/*!
*  \brief     Factory of the QXF
*  \author    Jean Iwanowski
*  \version   1.0
*
*  This class is responsible for the instanciation of all necessary classes for
*  the QXF to run properly. It is run by Qt's main thread.
*  It creates a QxfThread for the time management, and as many QxfThread as
*  necessary to run all the active state machines of a specific model.
*  The QxfTimeoutManager, all the QxfEventDispatcher and QxfReactive are also
*  instantiated in this Factory  so they can be linked together. Then they are
*  pushed into their respective threads, for pseudo-parallel execution.
*/
class QxfFactory : public QObject
{
private:
    QThread* pTimeoutThread;
    QThread* pReactiveThread;
    QThread* pReactiveThread2;
    QxfTimeoutManager * pTM;
    QxfEventDispatcher * pED;
    QxfEventDispatcher * pED2;
    QxfReactive * pR;
    QxfReactive * pR2;

public:
    // constructors and destructor
    QxfFactory();
    ~QxfFactory();
};

#endif // QXFFACTORY_H
```

Fichier de definitions

```
#include "qxffactory.h"

QxfFactory::QxfFactory()
{
    // creation of the threads
    pTimeoutThread = new QThread();
    pReactiveThread = new QThread();
    pReactiveThread2 = new QThread();

    // creation of the timeout manager
    pTM = new QxfTimeoutManager( 100 );

    // creation of the event dispatchers
    pED = new QxfEventDispatcher();
    pED2 = new QxfEventDispatcher();

    // creation of the state machines
    pR = new SimpleStateMachine1( pED, pTM );
    pR2 = new SimpleStateMachine2( pED2, pTM );

    // initialization of the relations between state machines
    pR->initRelations(pR2);
    pR2->initRelations(pR);

    // start threads
    pTimeoutThread->start( QThread::TimeCriticalPriority );
    pReactiveThread->start( QThread::NormalPriority );
    pReactiveThread2->start( QThread::NormalPriority );

    // push classes to the right threads
    pTM->moveToThread( pTimeoutThread );
    pED->moveToThread( pReactiveThread );
    pR->moveToThread( pReactiveThread );
    pED2->moveToThread( pReactiveThread2 );
    pR2->moveToThread( pReactiveThread2 );

    // start state machines
    pR->startBehavior();
    pR2->startBehavior();
}

QxfFactory::~QxfFactory()
{
    delete pTimeoutThread;
    delete pReactiveThread;
    delete pReactiveThread2;
}
```

12. Machines de test du QXF

Fichier d'en-têtes de la première machine

```
#ifndef SimpleStateMachine1_H
#define SimpleStateMachine1_H

#include<qxfreactive.h>
#include <QtDebug>
#include <testevent.h>
#include <qxfdefaultevent.h>
#include <qxftimeoutevent.h>
#include <qxfstartbehaviorevent.h>

class SimpleStateMachine1 : public QxfReactive
{
private :
    // enum of states in the state machine
    typedef enum states
    {
        ST_START,
        ST_A,
        ST_B
    }state;

    // enum of transitions in the state machine
    typedef enum transitions
    {
        TR_START_BEHAVIOR,
        TR_A_TO_B,
        TR_B_TO_A
    }transition;

     // enum of events in the state machine
    typedef enum events
    {
        EV_DEFAULT,
        EV_START_BEHAVIOR,
        EV_TIMEOUT,
        EV_TEST
    }event;

    // remembers the current state
    state currentState;
    state oldState;
    transition currentTransition;
    bool hasChanged;

public:
    // constructors and destructors
    SimpleStateMachine1(QxfEventDispatcher* pEventDispatcher,
QxfTimeoutManager* pTimeoutManager);

    /*!
    *   Defines the beahviour of the state machine.
    */
    void behavior(QxfEvent* pEvent);

    /*!
    *   Links the state machine with others state machines.
    *   \param pReactive pointer on the state machine to link.
    */
    void initRelations(QxfReactive* pReactive);

    /*!
    *   Finds the type of a subclass of QxfEvent.
```

```
    *   \param pEvent pointer on the event to analyse.
    *   \return numeric code of the type.
    */
    int typeOf(QxfEvent* pEvent);
};

#endif // SimpleStateMachine1_H
```

Fichier de définitions de la première machine

```
#include "simplestatemachine1.h"

SimpleStateMachine1::SimpleStateMachine1(QxfEventDispatcher* pEventDispatcher,
                QxfTimeoutManager* pTimeoutManager
                ):QxfReactive(pEventDispatcher, pTimeoutManager)
{
    // attributes init
    oldState = (state)0;
    currentState = (state)0;
    currentTransition = (transition)0;
    hasChanged = false;
}

void SimpleStateMachine1::initRelations(QxfReactive* pReactive)
{
}

int SimpleStateMachine1::typeOf(QxfEvent* pEvent)
{
    if( dynamic_cast<QxfStartBehaviorEvent*>( pEvent ) )
    {
        return EV_START_BEHAVIOR;
    }
    else if( dynamic_cast<QxfTimeoutEvent*>( pEvent ) )
    {
        return EV_TIMEOUT;
    }
    else if( dynamic_cast<QxfDefaultEvent*>( pEvent ) )
    {
        return EV_DEFAULT;
    }
    else if( dynamic_cast<TestEvent*>( pEvent ) )
    {
        return EV_TEST;
    }
    else return -1;
}

void SimpleStateMachine1::behavior(QxfEvent* pEvent)
{
   // Define and initialize local variables
    oldState = currentState;
    hasChanged = false;

    // Compute subsequent state and transition
    switch(currentState)
    {
        case ST_START:
            //check guards/events/timers if available
            if( typeOf( pEvent ) == EV START BEHAVIOR )
            {
                // set transition
                currentTransition = TR_START_BEHAVIOR;
                // set subsequent state
                currentState = ST A;
                // indicate changes
                hasChanged = true;
```

```
                qDebug() << "Enter : SM1 A";
            }
            break;

        case ST_A:
            //check guards/events/timers if available
            if(typeOf( pEvent ) == EV TEST )
            {
                // set transition
                currentTransition = TR_A_TO_B;
                // set subsequent state
                currentState = ST B;
                // indicate changes
                hasChanged = true;
                qDebug() << "Enter : SM1 B";
            }
            break;

        case ST B:
            //check guards/events/timers if available
            if(typeOf(pEvent) == EV TIMEOUT)
            {
                // set transition
                currentTransition = TR B TO A;
                // set subsequent state
                currentState = ST A;
                // indicate changes
                hasChanged = true;
                qDebug() << "Enter : SM1 A";
            }
            break;
    }

    // Start and push timers as well as default events
    if(hasChanged == true)
    {
        // removing obsolete timers
        emit stateChanged(this);

        switch(currentState)
        {
            case ST START:
                // generate and push default event if available
                // generate and push timer event if available
                break;

            case ST_A:
                // generate and push default event if available
                // generate and push timer event if available
                break;

            case ST_B:
                // generate and push default event if available
                // generate and push timer event if available
                createTimeout( 10 );
                break;
         }
    }

    // execute actions
    if(hasChanged == true)
    {
        // Execute exit actions of the previous state
        switch(oldState)
        {
            case ST START:
                // no action to perform as it is state machine entry
                break;
             case ST A:
                // execute actions
```

```
                break;

            case ST B:
                 // execute actions
                break;
          }

        // Execute actions on the current transition
        switch(currentTransition)
        {
            case TR_START_BEHAVIOR:
                // no action to perform as it is state machine entry transition
                break;

            case TR A TO B:
                // execute actions
                break;

            case TR B TO A:
                // execute actions
                 break;
         }

        // Execute entry actions of the current state
        switch(currentState)
        {
            case ST START:
                // no action to perform as it is state machine entry
                break;

            case ST A:
                // execute actions
                break;

            case ST_B:
                 // execute actions
                break;
         }
    }
}
```

Fichier d'en-têtes de la deuxième machine

```
#ifndef SimpleStateMachine2 H
#define SimpleStateMachine2 H

#include<qxfreactive.h>
#include <QtDebug>
#include <testevent.h>
#include <qxfdefaultevent.h>
#include <qxftimeoutevent.h>
#include <qxfstartbehaviorevent.h>

class SimpleStateMachine2 : public QxfReactive
{
private :
    // enum of states in the state machine
    typedef enum states
    {
        ST_START,
        ST_A
    }state;

    // enum of transitions in the state machine
    typedef enum transitions
    {
        TR_START_BEHAVIOR,
        TR_A_TO_A
```

```
    }transition;

     // enum of events in the state machine
    typedef enum events
    {
        EV_DEFAULT,
        EV_START_BEHAVIOR,
        EV_TIMEOUT,
        EV_TEST
    }event;

    // remembers the current state
    state currentState;
    state oldState;
    transition currentTransition;
    bool hasChanged;

    // pointers on related state machines
    QxfReactive* pSimpleStateMachine1;

public:
    // constructors and destructors
    SimpleStateMachine2(QxfEventDispatcher* pEventDispatcher,
QxfTimeoutManager* pTimeoutManager);

    /*!
  *  Defines the beahvior of the state machine.
  */
    void behavior(QxfEvent* pEvent);

    /*!
  *  Links the state machine with others state machines.
  *  \param pReactive pointer on the state machine to link.
  */
    void initRelations(QxfReactive* pReactive);

    /*!
  *  Finds the type of a subclass of QxfEvent.
  *  \param pEvent pointer on the event to analyse.
  *  \return numeric code of the type.
  */
    int typeOf(QxfEvent* pEvent);
};

#endif // SimpleStateMachine2_H
```

Fichier de définitions de la deuxième machine

```
#include "simplestatemachine2.h"

SimpleStateMachine2::SimpleStateMachine2(QxfEventDispatcher* pEventDispatcher,
                QxfTimeoutManager* pTimeoutManager
                ):QxfReactive(pEventDispatcher, pTimeoutManager)
{
    // attributes init
    oldState = (state)0;
    currentState = (state)0;
    currentTransition = (transition)0;
    hasChanged = false;

    pSimpleStateMachine1 = 0;
}

void SimpleStateMachine2::initRelations(QxfReactive* pReactive)
```

```
{
    pSimpleStateMachine1 = pReactive;
}

int SimpleStateMachine2::typeOf(QxfEvent* pEvent)
{
    if( dynamic_cast<QxfStartBehaviorEvent*>( pEvent ) )
    {
        return EV_START_BEHAVIOR;
    }
    else if( dynamic_cast<QxfTimeoutEvent*>( pEvent ) )
    {
        return EV_TIMEOUT;
    }
    else if( dynamic_cast<QxfDefaultEvent*>( pEvent ) )
    {
        return EV_DEFAULT;
    }
    else if( dynamic_cast<TestEvent*>( pEvent ) )
    {
        return EV_TEST;
    }
    else return -1;
}

void SimpleStateMachine2::behavior(QxfEvent* pEvent)
{
   // Define and initialize local variables
    oldState = currentState;
    hasChanged = false;

    // Compute subsequent state and transition
    switch(currentState)
    {
        case ST START:
            //check guards/events/timers if available
            if( typeOf( pEvent ) == EV START BEHAVIOR )
            {
                // set transition
                currentTransition = TR_START_BEHAVIOR;
                // set subsequent state
                currentState = ST A;
                // indicate changes
                hasChanged = true;
                qDebug() << "Enter : SM2 A";
            }
            break;

        case ST A:
            //check guards/events/timers if available
            if(typeOf( pEvent ) == EV_TIMEOUT )
            {
                // set transition
                currentTransition = TR A TO A;
                // set subsequent state
                currentState = ST A;
                // indicate changes
                hasChanged = true;
                qDebug() << "Enter : SM2 A";
            }
            break;
    }

    // Start and push timers as well as default events
    if(hasChanged == true)
    {
        // removing obsolete timers
        emit stateChanged(this);

        switch(currentState)
```

```
        {
            case ST START:
                // generate and push default event if available
                // generate and push timer event if available
                break;

            case ST A:
                // generate and push default event if available
                // generate and push timer event if available
                createTimeout( 50 );
                break;
         }
    }

    // execute actions
    if(hasChanged == true)
    {
        // Execute exit actions of the previous state
        switch(oldState)
        {
            case ST START:
                // no action to perform as it is state machine entry
                break;
             case ST A:
                // execute actions
                break;
          }

        // Execute actions on the current transition
        switch(currentTransition)
        {
            case TR START BEHAVIOR:
                // no action to perform as it is state machine entry transition
                break;

            case TR A TO A:
                // execute actions
                break;
         }

        // Execute entry actions of the current state
        switch(currentState)
        {
            case ST START:
                // no action to perform as it is state machine entry
                break;

            case ST A:
                // execute actions
                qDebug() << "----------------------";
                qDebug() << "Send TestEvent";
                qDebug() << "----------------------";
                GEN( TestEvent, *pSimpleStateMachine1 );
                break;
         }
    }
}
```

13. Fichier « Generator.mqr »

```
package rules;

public ruleset Generator(in model : uml21) {

	public rule main()
	{
		$Project(model);
		$Main();
		$StateMachineCommon();
		$FactoryHeader(model);
		$FactoryBody(model);
		foreach( stateMachine : uml21.Class
		in model.getInstances( "Class" ).select( "hasStateMachine", true )
)
		{
			$StateMachineHeader(stateMachine, model);
			$StateMachineBody(stateMachine, model);
		}
	}
}
```

14. Fichier « Main.tgt »

```
[#package rules]
[#template public Main()]
[#file]generated/main.cpp[/#file]
#include <QtCore/QCoreApplication>
#include "Factory.h"

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    Factory theFactory;
    return a.exec();
}
[/#template]
```

15. Fichier « Project.tgt »

```
[#package rules]
[#import com.sodius.mdw.core.model.*]
[#template public Project(in UML : uml21)]
[#set stateMachines  : MDWList      = UML.getInstances( "StateMachine" )
]
[#file]generated/SCEdit.pro[/#file]
QT       -= gui
TARGET = SCEdit
CONFIG   += console
CONFIG   -= app bundle
TEMPLATE = app

HEADERS += [#foreach stateMachine in
stateMachines]${stateMachine.getClass.name}.h \
[/#foreach]
      factory.h\
      StateMachineCommon.h

SOURCES += [#foreach stateMachine in
stateMachines]${stateMachine.getClass.name}.cpp \
[/#foreach]
      factory.cpp \
      main.cpp
[/#template]
```

16. Fichiers « Factory »

Fichier « FactoryHeader.tgt »

```
[#package rules]
[#import com.sodius.mdw.core.model.*]

[#template public FactoryHeader( in model : uml21 )]
[#set instances                              : MDWList      = model.getInstances(
"InstanceSpecification" )
              classInstances                 : MDWList      = instances.select(
"isClass", true )
              associationInstances   : MDWList      = instances.select(
"isAssociation", true )
              stateMachineInstances  : MDWList      = classInstances.select(
"hasStateMachine", true )
              stateMachineClasses            : MDWList      =
stateMachineInstances.classifier
              activeInstances                        : MDWList      =
stateMachineInstances.select("isActive", true)
]
[#file]generated/Factory.h[/#file]
#ifndef Factory H
#define Factory H

// Qt
#include <QObject>
#include <QStateMachine>
#include <QState>
#include <QFinalState>
#include <QThread>

// classes representing statemachines
[#foreach class in stateMachineClasses]
#include "${class.name}.h"
[/#foreach]

class Factory : public QObject
{
Q OBJECT
private:
    [#foreach instance in activeInstances]
    QThread TD_${instance.name};
    [/#foreach]

    [#foreach instance in stateMachineInstances]
    ${instance.classifier.first().name} SM ${instance.name};
    [/#foreach]

public:
    explicit Factory(QObject *parent = 0);
};

#endif // Factory_H
[/#template]
```

Fichier « FactoryBody.tgt »

```
[#package rules]
[#import com.sodius.mdw.core.model.*]

[#template public FactoryBody( in model : uml21 )]
[#file]generated/Factory.cpp[/#file]
[#set instances                              : MDWList      = model.getInstances(
"InstanceSpecification" )
```

```
            classInstances              : MDWList      = instances.select(
"isClass", true )
            associationInstances   : MDWList      = instances.select(
"isAssociation", true )
            stateMachineInstances  : MDWList      = classInstances.select(
"hasStateMachine", true )
            stateMachineClasses         : MDWList      =
stateMachineInstances.classifier
            activeInstances                     : MDWList      =
stateMachineInstances.select("isActive", true)

            states                              : MDWList      =
model.getInstances( "State" )
            transitions                         : MDWList      =
model.getInstances( "Transition" ).select("isInitial", false)
            stateMachines               : MDWList      = model.getInstances(
"StateMachine" )
            orthogonalStateMachines     : MDWList      =
stateMachines.select( "isOrthogonal", true)
            concurrentRegions           : MDWList      = model.getInstances(
"Region" ).select( "isConcurrent", true )
            orthogonalStates            : MDWList      = states.select(
"isOrthogonal", true)
            initialStates               : MDWList      = states.select(
"isInitial", true)
            finalStates                         : MDWList      =
model.getInstances( "FinalState" )
            activeClasses               : MDWList      =
stateMachineClasses.select("isActive", true)
            passiveClasses              : MDWList      =
stateMachineClasses.select("isActive", false)
]
#include "Factory.h"

Factory::Factory(QObject *parent) :
    QObject(parent)
{
      // links instanciation
    [#-- only two ways links are implemented --]
    [#foreach instance in associationInstances]

      SM ${instance.slot[0].value.first().instance.name}.setSM ${instance.slot[
1].definingFeature.name}( &SM ${instance.slot[1].value.first().instance.name}
);

      SM_${instance.slot[1].value.first().instance.name}.setSM_${instance.slot[
0].definingFeature.name}( &SM_${instance.slot[0].value.first().instance.name}
);
    [/#foreach]

      // threads activation
    [#foreach instance in activeInstances]
      TD ${instance.name}.start();
    [/#foreach]

      // state machines affiliation to threads
    [#foreach instance in activeInstances]
    SM_${instance.name}.moveToThread( &TD_${instance.name} );
    [/#foreach]

      // state machines start
    [#foreach instance in stateMachineInstances]
    SM_${instance.name}.start();
    [/#foreach]
}
[/#template]
```

17. Fichier « StateMachineCommon.tgt »

```
[#package rules]

[#template public StateMachineCommon()]
[#file]generated/StateMachineCommon.h[/#file]
#define GEN(target, event)   ( getMachine()->getSM_##target()->emit_##event() )
[/#template]
```

18. Fichiers « StateMachine »

Fichier « StateMachineHeader.tgt »

```
[#package rules]
[#import com.sodius.mdw.core.model.*]
[#template public StateMachineHeader( class : uml21.Class, in model : uml21 )]
[#set relatedClasses                        : MDWList      = model.getInstances(
"Association" ).getClassesRelatedTo(class)
              elements                                     : MDWList      =
class.allOwnedElements()
              states                                       : MDWList      =
elements.getInstances( "State", true )
              transitions                                  : MDWList      =
elements.getInstances( "Transition" ).select("isInitial", false)
              triggeredTransitions          : MDWList      = transitions.select(
"isTriggered", true )
              signalTriggeredTransitions    : MDWList      =
triggeredTransitions.select( "isSignalTriggered", true )
              defaultTransitions                    : MDWList      =
transitions.select( "isTriggered", false )
              signalTriggers                        : MDWList      =
signalTriggeredTransitions.trigger
              concurrentRegions                     : MDWList      =
elements.getInstances( "Region" ).select( "isConcurrent", true )
              orthogonalStates                      : MDWList      =
states.select( "isOrthogonal", true)
              initialStates                         : MDWList      =
states.select( "isInitial", true)
              finalStates                                  : MDWList      =
elements.getInstances( "FinalState" )
]
[#file]generated/${class.name}.h[/#file]
#ifndef ${class.getClass.name} H
#define ${class.getClass.name} H

/*************************************************************
*                       - INCLUSIONS -                       *
*************************************************************/
#include <QObject>
#include <QStateMachine>
#include <QState>
#include <QFinalState>
#include <QSignalTransition>
#include <QTimer>
#include <QDebug>
#include "StateMachineCommon.h"

[#foreach state in states]
class ${state.pathName};
[/#foreach]
[#foreach region in concurrentRegions]
class ${region.pathName};
[/#foreach]
[#foreach transition in triggeredTransitions]
class ${transition.pathName};
[/#foreach]
[#foreach relatedClass in relatedClasses]
class ${relatedClass.name};
[/#foreach]


/*************************************************************
*                  - STATE MACHINE CLASS -                   *
*************************************************************/
class ${class.name} : public QStateMachine
```

```
{
Q_OBJECT
private:
    [#foreach state in states]
    ${state.pathName}* pST_${state.pathName};
    [/#foreach]

      [#foreach region in concurrentRegions]
    ${region.pathName}* pST_${region.pathName};
    [/#foreach]

    [#foreach relatedClass in relatedClasses]
    ${relatedClass.name}* pSM ${relatedClass.name};
    [/#foreach]

signals:
[#foreach trigger in signalTriggers]
      void ${trigger.event.name}();
[/#foreach]

public:
    explicit ${class.name}( QObject* parent = 0);

    ${class.name}* getMachine();

      [#foreach trigger in signalTriggers]
    void emit ${trigger.event.name}();
    [/#foreach]

    [#foreach relatedClass in relatedClasses]
     void setSM ${relatedClass.name}(${relatedClass.name}* sm);
     ${relatedClass.name}* getSM ${relatedClass.name}();
    [/#foreach]
};

/*************************************************************
*                      - STATE CLASSES -                     *
*************************************************************/
[#foreach state in states]
[#include StateHeader( state )]

[/#foreach]
[#foreach region in concurrentRegions]
[#include RegionHeader( region )]

[/#foreach]

/*************************************************************
*                    - TRANSITION CLASSES -                          *
*************************************************************/
[#foreach transition in triggeredTransitions]
[#include TransitionHeader( transition )]

[/#foreach]

#endif // ${class.name}_H
[/#template]
```

Fichier « StateMachineHeader.tgt »

```
[#package rules]
[#import com.sodius.mdw.core.model.*]
[#template public StateMachineBody( class : uml21.Class, in model : uml21 )]
[#set relatedClasses                        : MDWList      = model.getInstances(
"Association" ).getClassesRelatedTo(class)
             elements                                      : MDWList      =
class.allOwnedElements()
             states                                        : MDWList      =
elements.getInstances( "State", true )
             transitions                                   : MDWList      =
elements.getInstances( "Transition" ).select("isInitial", false)
             triggeredTransitions          : MDWList      = transitions.select(
"isTriggered", true )
             signalTriggeredTransitions    : MDWList      =
triggeredTransitions.select( "isSignalTriggered", true )
             defaultTransitions                    : MDWList      =
transitions.select( "isTriggered", false )
             signalTriggers                        : MDWList      =
signalTriggeredTransitions.trigger
             concurrentRegions                     : MDWList      =
elements.getInstances( "Region" ).select( "isConcurrent", true )
             orthogonalStates                      : MDWList      =
states.select( "isOrthogonal", true)
             initialStates                         : MDWList      =
states.select( "isInitial", true)
             finalStates                                   : MDWList      =
elements.getInstances( "FinalState" )
             stateMachine                          : uml21.StateMachine =
class.classifierBehavior
]
[#file]generated/${class.name}.cpp[/#file]
#include "${class.name}.h"
[#foreach relatedClass in relatedClasses]
#include "${relatedClass.name}.h"
[/#foreach]

${class.name}::${class.name}( QObject* parent) :
      QStateMachine(parent)
{
      /*****************************************
      * class instanciations                   *
      *****************************************/
    [#foreach state in states]
    pST ${state.pathName} = new ${state.pathName}();
    [/#foreach]

   [#foreach region in concurrentRegions]
    pST ${region.pathName} = new ${region.pathName}();
    [/#foreach]

      /*****************************************
      * specification of orthogonal states     *
      *****************************************/
      // orthogonal states
      [#foreach state in orthogonalStates]
    ST ${state.pathName}setChildMode( QState::ParallelStates );
      [/#foreach]

      //orthogonal state machines
      [#if stateMachine.isOrthogonal]
      this->setChildMode( QState::ParallelStates );
      [/#if]

      /*****************************************
      * addition of states to state machine    *
      *****************************************/
      // states
```

```
      [#foreach state in states]
    [#if state.getParent == stateMachine]
    pST ${state.pathName}->setParent( this );
    [#else]
    pST_${state.pathName}->setParent( pST_${state.getParent.pathName} );

    [/#if]
      [/#foreach]

      // states representing concurrent regions
      [#foreach region in concurrentRegions]
    [#if region.getParent == stateMachine]
      pST ${region.pathName}->setParent( this );
    [#else]
    pST ${region.pathName}->setParent( pST ${region.getParent.pathName} );

    [/#if]
      [/#foreach]

      /*****************************************
      * definition of the initial states       *
      *****************************************/
      [#foreach state in initialStates]
    [#if state.getParent == stateMachine]
    this->setInitialState( pST ${state.pathName} );
    [#else]
    pST ${state.getParent.pathName}->setInitialState( pST ${state.pathName} );
      [/#if]
      [/#foreach]

      /*****************************************
      * transitions creation                   *
      *****************************************/
      // default transitions
      [#foreach transition in defaultTransitions]
    pST ${transition.source.pathName}->addTransition(
pST ${transition.target.pathName} );
      [/#foreach]

      // triggered transitions
      [#foreach transition in triggeredTransitions]
    [#foreach trigger in transition.trigger]
      [#if trigger.event.eClass().name == "SignalEvent"]
      ${transition.pathName} TR ${transition.pathName}( this, SIGNAL(
${trigger.event.name}() ), pST ${transition.target.pathName} );
      pST_${transition.source.pathName}->addTransition(
&TR_${transition.pathName} );
      [/#if]
      [#if trigger.event.eClass().name == "TimeEvent"]
      ${transition.pathName} TR ${transition.pathName}(
pST_${transition.source.pathName}->getTimer(), SIGNAL( timeout() ),
pST_${transition.target.pathName} );
      pST ${transition.source.pathName}->addTransition(
&TR ${transition.pathName} );
      [/#if]
      [/#foreach]
    [/#foreach]
}

${class.name}* ${class.name}::getMachine()
{
      return this;
}


[#foreach trigger in signalTriggers]
void ${class.name}::emit ${trigger.event.name}()
{
      emit ${trigger.event.name}();
}
```

```
[/#foreach]

[#foreach relatedClass in relatedClasses]
void ${class.name}::setSM_${relatedClass.name}(${relatedClass.name}* sm)
{
      pSM ${relatedClass.name} = sm;
}

${relatedClass.name}* ${class.name}::getSM_${relatedClass.name}()
{
      return static_cast<${relatedClass.name}*>( pSM_${relatedClass.name} );
}
[/#foreach]

/*************************************************************
*                      - STATE CLASSES -                     *
*************************************************************/
[#foreach state in states]
[#include StateBody( state )]

[/#foreach]
[#foreach region in concurrentRegions]
[#include RegionBody( region )]

[/#foreach]

/*************************************************************
*                    - TRANSITION CLASSES -                  *
*************************************************************/
[#foreach transition in triggeredTransitions]
[#include TransitionBody( transition )]

[/#foreach]
[/#template]
```

19. Fichiers « State »

Fichier « StateHeader.tgt »

```
[#package rules]
[#template public StateHeader( state :uml21.State)]
#ifndef ${state.pathName} H
#define ${state.pathName}_H

class ${state.pathName} : public QState
{
Q OBJECT
public:
    explicit ${state.pathName}(QState *parent = 0);

        ${state.getClass.name}* getMachine();

    [#if state.getTimedTransition != null]
      QTimer* getTimer();
      [/#if]

protected:
    void onEntry( QEvent* event );
    void onExit( QEvent* event );

private:
[#if state.getTimedTransition != null]
      QTimer* pTimer;
[/#if]
};

#endif // ${state.pathName}_H
[/#template]
```

Fichier « StateBody.tgt »

```
[#package rules]
[#import com.sodius.mdw.core.model.*]
[#template public StateBody( state :uml21.State)]
[#set triggeredTransitions   : MDWList      = state.outgoing.select(
"isTriggered", true )
              triggers                              : MDWList      =
triggeredTransitions.trigger
]
${state.pathName}::${state.pathName}(QState *parent) :
    QState(parent)
{
    [#if state.getTimedTransition != null]
      pTimer = new QTimer(this);
      pTimer->setSingleShot(true);
      pTimer->setInterval(${state.getTimedTransition});
      [/#if]
}

[#if state.getTimedTransition != null]
QTimer* ${state.pathName}::getTimer()
{
      return pTimer;
}
[/#if]

${state.getClass.name}* ${state.pathName}::getMachine()
{
      return ( (${state.getClass.name}*)machine() )->getMachine();
}

void ${state.pathName}::onEntry( QEvent* event )
{
    qDebug() << "Entry : ${state.pathName}";
    [#if state.getTimedTransition != null]
      pTimer->start();
      [/#if]
      [#if state.entry.eClass().name == "OpaqueBehavior"]
    ${state.entry.concat}
    [/#if]
}

void ${state.pathName}::onExit( QEvent* event )
{
    qDebug() << "Exit  : ${state.pathName}";
    [#if state.getTimedTransition != null]
      pTimer->stop();
      [/#if]
    [#if state.entry.eClass().name == "OpaqueBehavior"]
    ${state.exit.concat}
    [/#if]
}
[/#template]
```

20. Fichiers « Region »

Fichier « RegionHeader.tgt »

```
[#package rules]
[#template public RegionBody( region :uml21.Region)]
${region.pathName}::${region.pathName}(QState *parent) :
    QState(parent)
{
}

void ${region.pathName}::onEntry( QEvent* event )
{
}

void ${region.pathName}::onExit( QEvent* event )
{
}
[/#template]
```

Fichier « RegionBody.tgt »

```
[#package rules]
[#template public RegionHeader( region :uml21.Region)]
#ifndef ${region.pathName} H
#define ${region.pathName} H

class ${region.pathName} : public QState
{
Q OBJECT
public:
    explicit ${region.pathName}(QState *parent = 0);

protected:
    void onEntry( QEvent* event );
    void onExit( QEvent* event );
};

#endif // ${region.pathName} H
[/#template]
```

21. Fichiers « Transition »

Fichier « TransitionHeader.tgt »

```
[#package rules]

[#template public TransitionHeader( transition : uml21.Transition )]
#ifndef ${transition.pathName}_H
#define ${transition.pathName}_H

class ${transition.pathName} : public QSignalTransition
 {
 public:
     ${transition.pathName}( QObject* sender, const char* signal, QState*
sourceState );

 protected:
      bool eventTest( QEvent *e );
      void onTransition( QEvent* e );
 };
 #endif // ${transition.pathName}_H
 [/#template]
```

Fichier « TransitionHeader.tgt »

```
[#package rules]
[#template public TransitionBody( transition : uml21.Transition )]
[#set test    : uml21.Element        = transition.guard.specification
]
${transition.pathName}::${transition.pathName}( QObject* sender, const char*
signal, QState* sourceState = 0 )
     : QSignalTransition( sender, signal, sourceState )
 {
 }

void ${transition.pathName}::onTransition( QEvent* e )
{
    qDebug() << "Trans : ${transition.pathName}";
    [#if transition.eIsInstanceOf("Transition")]
    ${transition.effect.concat}
    [/#if]
}


bool ${transition.pathName}::eventTest(QEvent *e)
 {
     if (!QSignalTransition::eventTest(e))
     {
         return false;
     }
     [#if !( test == null )]
     [#if test.eClass().name == "OpaqueExpression"]
     if(!${test.body.first()})
     {
      return false;
     }
     [/#if]
     [/#if]
     return true;
 }
[/#template]
```

22. Fichier « uml21_Association.mqs »

```
package rules;
import com.sodius.mdw.core.model.*;
metatype uml21.Association;

public script getClassesRelatedTo( class : uml21.Class) : MDWList {
	var list : MDWList = [];
	var related : boolean = false;

	foreach(end in self.endType)
	{
		if(	end != class )
		{
			list.add(end);
		}
		else
		{
			related = true;
		}
	}
	if(	related )
	{
		return list;
	}
	return null;
}
```

23. Fichier « uml21_Class.mqs »

```
package rules;

metatype uml21.Class;

public script hasStateMachine() : boolean
{
	return( self.classifierBehavior.eClass().name == "StateMachine" );
}
```

24. Fichier « uml21_Element.mqs »

```
package rules;
metatype uml21.Element;

public script pathName() : String {
	var myElement : uml21.Element = self;
	var pathName : String = myElement.name;
	while( myElement.eClass().name != "Class" )
	{
		myElement = myElement.owner;
		if( myElement.eIsInstanceOf( "State" ) )
		{
			var temp : uml21.State = myElement;
			pathName = temp.name + "_" + pathName;
		}
		else if( myElement.eClass().name == "Region" )
		{
			var temp : uml21.Region = myElement;
			pathName = temp.name + " " + pathName;
		}
		else if( myElement.eClass().name == "Class" )
		{
			var temp : uml21.Class = myElement;
			pathName = temp.name + " " + pathName;
		}
	}
	return pathName;
}

public script getParent() : uml21.Element
{
	var myElement : uml21.Element = self.owner;
	while( myElement.eClass().name != "StateMachine" )
	{
		if( myElement.eIsInstanceOf( "State" ) )
		{
			return myElement;
		}
		else if( myElement.eClass().name == "Region" )
		{
			if(myElement.isConcurrent)
			{
				return myElement;
			}
		}
		myElement = myElement.owner;
	}
	return myElement;
}

public script getClass() : uml21.Class
{
	var myElement : uml21.Element = self;
	while( myElement.eClass().name != "Class" )
	{
		myElement = myElement.owner;
	}
	if( myElement.eClass().name == "Class" )
	{
		var temp : uml21.Class = myElement;
		return temp;
	}
}
```

25. Fichier « uml21_InstanceSpecification.mqs »

```
package rules;
metatype uml21.InstanceSpecification;

public script isClass() {
	return (self.classifier.first().eClass().name == "Class");
}

public script isAssociation() {
	return (self.classifier.first().eClass().name == "Association");
}

public script isActive() {
	if(self.isClass)
	{
		return (self.classifier.first().isActive);
	}
	return false;
}

public script hasStateMachine() {
	if(self.isClass)
	{
		return self.classifier.first().hasStateMachine;
	}
	return false;
}
```

26. Fichier « uml21_Behavior.mqs »

```
package rules;

metatype uml21.OpaqueBehavior;

public script concat() {
	var str : String;
	foreach (string : String in self.body) {
		str = str + "\n\t\t\t\t" + string;
	}
	return str.substring(5, str.length());
}
```

27. Fichier « uml21_Region.mqs »

```
package rules;

metatype uml21.Region;

public script isConcurrent() : boolean
{
	return ( self.owner.ownedElement.getInstances( "Region" ).size() > 1 );
}
```

28. Fichier « uml21_State.mqs »

```
package rules;

metatype uml21.State;

public script isInitial() : boolean
{
	var ret : boolean = false;
	foreach (transition : uml21.Transition in self.incoming)
	{
		if( transition.isInitial )
		{
			ret = true;
		}
	}
	return ret;
}

public script getTimedTransition() : int
{
	var ret : int = null;
	foreach (event : uml21.Event in self.outgoing.trigger.event)
	{
		if( event.eClass().name == "TimeEvent" )
		{
			ret = event.when.value;
		}
	}
	return ret;
}
```

29. Fichier « uml21_StateMachine.mqs »

```
package rules;

metatype uml21.StateMachine;

public script isOrthogonal() : boolean
{
	return ( self.ownedElement.getInstances( "Region" ).size() > 1 );
}

public script getStatemachine() : uml21.StateMachine
{

}
```

30. Fichier « uml21_Transition.mqs »

```
package rules;

metatype uml21.Transition;

public script isInitial()
{
	var source : uml21.Element = self.source;
	if( source.eClass().name == "Pseudostate" )
	{
		var temp : uml21.Pseudostate = source;
		if( temp.kind.getName() == "initial" )
		{
			return true;
		}
	}
	return false;
}

public script isTriggered() : boolean
{
	return !( self.trigger.isEmpty() );
}

public script isSignalTriggered() : boolean
{
	if(self.isTriggered)
	{
		foreach( trigger : uml21.Trigger in self.trigger )
		{
			if( trigger.event.eClass().name == "TimeEvent")
			{
				return false;
			}
		}
		return true;
	}
	return false;
}

public script isTimeTriggered() : boolean
{
	if(self.isTriggered)
	{
		return !self.isSignalTriggered;
	}
	return false;
}
```