UNION INTERNATIONALE DES TÉLÉCOMMUNICATIONS

UIT-T

SECTEUR DE LA NORMALISATION
DES TÉLÉCOMMUNICATIONS
DE L'UIT

T.173

(07/97)

SÉRIE T: TERMINAUX DES SERVICES TÉLÉMATIQUES

Représentation des scripts MHEG-3 pour les échanges

Recommandation UIT-T T.173

(Antérieurement Recommandation du CCITT)

RECOMMANDATIONS UIT-T DE LA SÉRIE T

TERMINAUX DES SERVICES TÉLÉMATIQUES

Pour plus de détails, voir la Liste des Recommandations de l'UIT-T.

RECOMMANDATION UIT-T T.173

REPRESENTATION DES SCRIPTS MHEG-3 POUR LES ECHANGES

Source

La Recommandation UIT-T T.173, élaborée par la Commission d'études 16 (1997-2000) de l'UIT-T, a été approuvée le 10 juillet 1997 selon la procédure définie dans la Résolution n° 1 de la CMNT.

AVANT-PROPOS

L'UIT (Union internationale des télécommunications) est une institution spécialisée des Nations Unies dans le domaine des télécommunications. L'UIT-T (Secteur de la normalisation des télécommunications) est un organe permanent de l'UIT. Il est chargé de l'étude des questions techniques, d'exploitation et de tarification, et émet à ce sujet des Recommandations en vue de la normalisation des télécommunications à l'échelle mondiale.

La Conférence mondiale de normalisation des télécommunications (CMNT), qui se réunit tous les quatre ans, détermine les thèmes d'études à traiter par les Commissions d'études de l'UIT-T lesquelles élaborent en retour des Recommandations sur ces thèmes.

L'approbation des Recommandations par les Membres de l'UIT-T s'effectue selon la procédure définie dans la Résolution n° 1 de la CMNT.

Dans certains secteurs de la technologie de l'information qui correspondent à la sphère de compétence de l'UIT-T, les normes nécessaires se préparent en collaboration avec l'ISO et la CEI.

NOTE

Dans la présente Recommandation, l'expression "Administration" est utilisée pour désigner de façon abrégée aussi bien une administration de télécommunications qu'une exploitation reconnue.

DROITS DE PROPRIÉTÉ INTELLECTUELLE

L'UIT attire l'attention sur la possibilité que l'application ou la mise en œuvre de la présente Recommandation puisse donner lieu à l'utilisation d'un droit de propriété intellectuelle. L'UIT ne prend pas position en ce qui concerne l'existence, la validité ou l'applicabilité des droits de propriété intellectuelle, qu'ils soient revendiqués par un Membre de l'UIT ou par une tierce partie étrangère à la procédure d'élaboration des Recommandations.

A la date d'approbation de la présente Recommandation, l'UIT avait/n'avait pas été avisée de l'existence d'une propriété intellectuelle protégée par des brevets à acquérir pour mettre en œuvre la présente Recommandation. Toutefois, comme il ne s'agit peut-être pas de renseignements les plus récents, il est vivement recommandé aux responsables de la mise en œuvre de consulter la base de données des brevets du TSB.

TABLE DES MATIÈRES

		Page
1	Domaine	1
2	Références normatives	1
3	Définitions	2
3.1	Définitions	2
3.2	Abréviations	6
4	Généralités	7
5	Conformité	8
5.1	Conformité d'un objet d'information	8
5.1.1	Profils	8
5.1.2	Codage	9
5.1.3	Syntaxe	9
5.1.4	Sémantique	9
5.2	Conformité d'implémentation	9
5.2.1	Requêtes de conformité	9
5.2.2	Documentation de conformité	10
5.3	Conformité d'application	10
5.4	Méthodes des test	10
6	Aperçu général	10
6.1	Méthodologie de description	11
6.2	Opérations de traitement de données	11
6.3	Accès aux données et fonctions externes	12
7	Relations entre MHEG et MHEG-3	13
7.1	Entités MHEG	13
7.2	Entités fonctionnelles	13
7.3	Interpréteur de représentation d'échange de script (SIR MHEG)	13
8	Composants de la représentation de script pour les échanges MHEG (SIR MHEG)	14
8.1	Types de données	14
8.1.1	Types prédéfinis	14
8.1.2	Types construits déclarés	17
8.2	Données	19
8.2.1	Valeurs immédiates	20
8.2.2	Constantes	20
8.2.3	Variables	20

Page

8.3 Fonctions 21
8.3.1 Routines 22
8.3.2 Services 22
8.3.3 Fonctions prédéfinies 23
8.4 Messages 23
8.4.1 Exceptions de bloc 23
8.4.2 Messages prédéfinis 24
8.5 Instructions 24
8.6 Identificateurs 24
8.6.1 Identificateurs de type 24
8.6.2 Identificateurs de données 25
8.6.3 Identificateurs de fonctions 25
8.6.4 Identificateurs de messages 25
9 La machine virtuelle SIR MHEG 26
9.1 Structure de la machine virtuelle SIR MHEG 26
9.2 Structures et notations 26
9.2.1 Table 26
9.2.2 Pile 27
9.2.3 Pile de paramètres 27
9.2.4 File d'attente 27
9.2.5 Représentation des données 27
9.3 Zones de mémoire 29
9.3.1 Zone de mémoire de *mh-script* 29
9.3.2 Zones de mémoire du *rt-script* 32
9.4 Etats des scripts 35
9.4.1 Etat d'un mh-script 35
9.4.2 Etats d'un rt-script 35
9.5 Unités de traitement 36
9.5.1 Réception de message 37
9.5.2 Initialisation de mh-script 38
9.5.3 Initialisation de rt-script 38
9.5.4 Unité d'exécution de rt-script 38
9.5.5 Unité d'exécution d'instructions SIR MHEG 39
10 Dispositions pour l'accès à un environnement d'exécution 39
10.1 Modèle général 39
10.2 Déclaration des interfaces IDL 40
10.3 Invocation d'opérations externes dans un programme SIR MHEG 40
10.4 Manipulation d'exceptions externes dans un programme SIR MHEG 41

Page

10.5 Invocation d'opérations externes par un moteur MHEG-3 41
10.6 Manipulation des exceptions externes par un moteur MHEG-3 41
10.7 Spécifications de mappage de plate-forme 41

11 Dispositions pour la manipulation d'objets MHEG 42
11.1 Invocation d'action MHEG 42
11.1.1 Envoi de messages à d'autres scripts 42
11.1.2 Echange d'information avec des objets MHEG 43
11.2 Réception de messages MHEG 43
11.2.1 Opération run de l'API MHEG-3 43
11.2.2 Exceptions de l'API MHEG 43

12 Déclarations de la SIR MHEG 43
12.1 Déclaration de type 44
12.1.1 Identificateur de type 44
12.1.2 Description de type 44
12.2 Déclaration de constante 46
12.2.1 Identificateur de donnée 46
12.2.2 Identificateur de type 46
12.2.3 Valeur de constante 46
12.3 Déclaration de variable globale 47
12.3.1 Identificateur de donnée 47
12.3.2 Identificateur de type 47
12.3.3 Référence à constante 47
12.4 Déclaration de bloc 48
12.4.1 Identificateur de bloc 48
12.4.2 Nom 48
12.4.3 Description de service 48
12.4.4 Description d'exception 50
12.5 Déclaration de filet 50
12.5.1 Identificateur de message 51
12.5.2 Identificateur de fonction 51
12.6 Déclaration de routine 51
12.6.1 Identificateur de fonction 51
12.6.2 Identificateur de type 51
12.6.3 Description de paramètre 51
12.6.4 Déclaration de variable locale 52
12.6.5 Code programme 53

Page

13 Instructions de la SIR MHEG 53
13.1 Méthodologie de présentation 53
13.1.1 Conditions d'erreur 53
13.1.2 Spécification formelle 54
13.1.3 Notation de table de données (data table) 54
13.1.4 Notation d'instruction générique 54
13.1.5 Primitives 55
13.2 Classification des instructions de la SIR MHEG 55
13.3 Description des instructions 57
13.3.1 Pas d'opération (no operation) 57
13.3.2 Remise (Yield) 57
13.3.3 Retour (Return) 58
13.3.4 Libération (Free) 58
13.3.5 Non (Not) 59
13.3.6 Ou (Or) 59
13.3.7 Ou exclusif (Exclusive or) 60
13.3.8 Et (And) 60
13.3.9 Egalité de référence (Equal reference) 60
13.3.10 Egal (Equal) 61
13.3.11 Inférieur à (Less than) 61
13.3.12 Supérieur à (Greater than) 62
13.3.13 Addition (Add) 62
13.3.14 Soustraction (Subtract) 63
13.3.15 Multiplication (Multiply) 63
13.3.16 Division (Divide) 63
13.3.17 Négation (Negate) 64
13.3.18 Reste (Remainder) 64
13.3.19 Duplication (Duplicate) 65
13.3.20 Conversion (Convert) 65
13.3.21 Saut sur condition vraie (Jump on true) 65
13.3.22 Saut sur condition fausse (Jump on false) 66
13.3.23 Saut (Jump) 66
13.3.24 Décalage (Shift) 67
13.3.25 Extraction de références à objet (Get object reference) 67
13.3.26 Saut long sur condition vraie (Long jump on true) 67
13.3.27 Saut long sur condition fausse (Long jump on false) 68
13.3.28 Saut long (Long jump) 68
13.3.29 Appel (Call) 69
13.3.30 Appel externe (External call) 70
13.3.31 Empiler (Push) 71

Page

13.3.32 Empiler référence (Push reference) 72
13.3.33 Empiler valeur immédiate (Push immediate) 72
13.3.34 Dépiler (Pop) 73
13.3.35 Dépiler référence (Pop reference) 73
13.3.36 Dépiler contenus (Pop contents) 73
13.3.37 Allocation (Allocate) 74
13.3.38 Incrément (Increment) 74
13.3.39 Décrément (Decrement) 75
13.3.40 Extraction (Get) 75
13.3.41 Extraction de contenus (Get contents) 76
13.3.42 Affectation (Set) 77
13.3.43 Affectation de contenus (Set contents) 78
13.4 Règles de conversion de types 78
13.4.1 Conversions réversibles 79
13.4.2 Extensions sans perte 79
13.4.3 Extensions avec perte 80
13.4.4 Troncations vers booléens 80
13.4.5 Troncations entre types entiers ou flottants 80
13.4.6 Troncations de valeur flottante à entier 80
14 Mappage entre la SIR MHEG et IDL 80
14.1 Spécifications IDL 81
14.2 Modules et interfaces IDL 81
14.3 Opérations IDL 81
14.3.1 Nom d'opération 81
14.3.2 Paramètres d'opération 81
14.3.3 Paramètre implicite 81
14.3.4 Valeur de retour 82
14.4 Attributs IDL 82
14.4.1 Accesseur (Accessor) 82
14.4.2 Modificateur (Modifier) 82
14.4.3 Attribut en lecture seule 82
14.5 Opérations IDL héritées 82
14.6 Exceptions IDL 82
14.6.1 Nom d'exception 82
14.6.2 Membres d'une exception 82
14.6.3 Membre implicite 83
14.7 Types IDL 83
14.7.1 Type char 83
14.7.2 Type enum 83

Page

14.7.3 Types construits 84
14.7.4 Type any 84
14.7.5 Restrictions sur les types 84
14.8 Constantes IDL 84
15 L'API MHEG-3 84
15.1 Objet interpréteur de script (ScriptInterpreter) 85
15.1.1 Opération Détruire (kill) 85
15.1.2 Opération Préparer (prepare) 85
15.2 Objet MhScript 86
15.2.1 Opération détruire (destroy) 86
15.2.2 Opération new 86
15.3 Object RtScript 87
15.3.1 Opération Détruire (delete) 87
15.3.2 Opération Affectation de priorité (setPriority) 87
15.3.3 Opération Extraction de priorité (getPriority) 88
15.3.4 Opération Affectation de donnée (setData) 88
15.3.5 Opération Extraction de donnée (getData) 88
15.3.6 Opération Allouer (Allocate) 89
15.3.7 Opération Libérer (free) 89
15.3.8 Opération Arrêter (stop) 90
15.3.9 Opération Réinitialisation (reInit) 90
15.3.10 Opération Extraction de statut de RtScript (getRtScriptStatus) 91
15.3.11 Opération Ouvrir (open) 91
15.4 Objet Invocation de routine (RoutineInvocation) 91
15.4.1 Opération Fermer (close) 92
15.4.2 Attribut en lecture seule Identificateur de routine (routine_id) 92
15.4.3 Opération Affectation de paramètre (setParameter) 92
15.4.4 Opération Extraction de prototype (getPrototype) 93
15.4.5 Opération Fonctionner (run) 93
15.4.6 Opération Remise à zéro (reset) 94
15.4.7 Opération Extraction du statut d'invocation (getInvocationStatus) 94
Annexe A – Spécification ASN.1 des scripts échangés 95
Annexe B – Représentation codée des scripts échangés 98
B.1 Codage des scripts échangés 98
B.2 Codage du code programme 99
B.2.1 Instructions code opérateur 99
B.2.2 Instructions opérande 99

Page

Annexe C – Eléments prédéfinis de la SIR MHEG 104

C.1 Types prédéfinis 104
C.1.1 Types primitifs 104
C.1.2 Types de l'API MHEG 105
C.2 Fonctions prédéfinies 105
C.2.1 Opérations de l'API MHEG 105
C.2.2 Opérations de l'API MHEG-3 106
C.3 Messages prédéfinis 106
C.3.1 Opérations de l'API MHEG-3 106
C.3.2 L'exception InstructionExecutionError (erreur d'exécution d'instruction) 107
C.3.3 Exceptions de l'API MHEG-3 107
C.3.4 Exceptions de l'API MHEG 108

Annexe D – Formulaire de spécification de mappage de plate-forme IDL 108

Annexe E – Processus de définition de l'API MHEG 109
E.1 Cadre de définition de l'API générique 109
E.1.1 Eléments MHEG en entrée du processus de définition de l'API MHEG 109
E.1.2 Eléments IDL en sortie du processus de définition de l'API MHEG 110
E.1.3 Contraintes vis-à-vis du processus de définition de l'API MHEG 110
E.1.4 Structure générale de l'API MHEG 111
E.1.5 Définition de types de données IDL non objets 112
E.1.6 Définition d'interface IDL 117
E.1.7 Définition d'attribut IDL 117
E.1.8 Définition d'opération IDL 118
E.1.9 Définition d'exception IDL 121
E.2 Mappage entre l'API MHEG et la SIR MHEG 122

Annexe F – Spécification IDL de l'API MHEG-3 123

Annexe G – Relations avec les autres parties des Recommandations UIT-T de la série T.170 (et parties de l'ISO/CEI 13522) 124
G.1 Relations avec la Rec. UIT-T T.171 (et l'ISO/CEI 13522-1) 124
G.2 Relations avec la Rec. UIT-T T.172 (et l'ISO/CEI 13522-5) 125

Appendice I – Syntaxe de la SIR MHEG (notation EBNF) 126

Appendice II – Notation textuelle des scripts de la SIR MHEG 128

Appendice III – Entités MHEG 131
III.1 Objets MHEG 131
III.2 Mh-objects 132

Page

III.3 Rt-objects 132
III.4 Objets MHEG échangés 133

Appendice IV – Principales caractéristiques de la SIR MHEG 133
IV.1 Caractéristiques des applications utilisatrices 133
IV.1.1 Manipulation d'entités MHEG 133
IV.1.2 Calcul, manipulation de variables et contrôle de structure 134
IV.1.3 Contrôle de dispositifs externes 134
IV.1.4 Acquisition de données 134
IV.1.5 Accès à des données externes 134
IV.1.6 Accès à des services externes d'exécution arbitraires 134
IV.2 Caractéristiques fonctionnelles 134
IV.2.1 Opérations de traitement de données 135
IV.2.2 Accès à des fonctions et données externes 135
IV.3 Caractéristiques techniques 136
IV.3.1 Indépendance vis-à-vis du matériel 136
IV.3.2 Représentation sous forme définitive 136
IV.3.3 Compacité 137
IV.3.4 Facilité d'implémentation 137
IV.3.5 Efficacité dans l'interprétation 137
IV.3.6 Ouverture et extensibilité 137
IV.3.7 Caractère non révisable 137
IV.3.8 Dispositions pour l'échange temps-réel 138
IV.3.9 Validation sémantique à des fins de qualité de service 138
IV.3.10 Caractère vérifiable de la syntaxe (en relation avec les dangers de contamination) 138
IV.3.11 Représentation non propriétaire 138
IV.3.12 Traitement de script sécurisé 138

Introduction

La présente Recommandation, qui est équivalente à l'ISO/CEI 13522-3 est une Recommandation générique qui étend la représentation codée de la classe d'objets des scripts MHEG définie dans les Recommandations UIT-T de la série T.170, et notamment T.171 et T.172. La présente Recommandation spécifie la représentation MHEG d'échange de scripts (SIR MHEG) pour les contenus des objets scripts, à savoir le codage des composants constituant les données des scripts de la classe script de MHEG.

Les Recommandations UIT-T de la série T.170 comprennent les Recommandations approuvées et en projet suivantes:

– T.170 (1998), *Recommandation de cadrage général pour la série T.170.*

– T.171 (1996), *Protocoles pour les services audiovisuels interactifs: représentation codée des objets multimédias et hypermédias.*

– T.172 (1998), *MHEG-5 – Support pour applications interactives de niveau fondamental.*

– T.173 (1997), *Représentation des scripts MHEG-3 pour les échanges.*

– T.174 (1996), *Interface de programmation d'application pour le système MHEG-1.*

– T.175 (1998), *Interface de programmation d'application pour le système MHEG-5.*

– T.176 (1998), *Interface de programmation d'application pour la commande et le contrôle de support de stockage numérique.*

Certaines Recommandations UIT-T de la série T.170 sont équivalentes aux parties de l'ISO/CEI 13522. Cette norme comprend les parties suivantes regroupées sous le titre global: Technologies de l'information – Codage de l'information multimédia et hypermédia.

– Partie 1: notation de base (ASN.1);

– Partie 3: représentation d'échange des scripts MHEG;

– Partie 4: procédure d'enregistrement MHEG;

– Partie 5: support pour applications interactives de niveau fondamental;

– Partie 6: support pour applications interactives de niveau évolué.

Les Annexes A à G sont parties intégrantes de la présente Recommandation. Les Appendices I à IV sont donnés à titre d'information seulement.

Recommandation T.173

REPRESENTATION DES SCRIPTS MHEG-3 POUR LES ECHANGES

(Genève, 1997)

1 Domaine

L'objectif de la présente Recommandation est d'étendre la représentation codée des objets MHEG appartenant à la classe script définie dans la Rec. UIT-T T.171 (et l'ISO/CEI 13522-1) [5] et dans la Rec. UIT-T T.172 (et l'ISO/CEI 13522-5) [7].

La présente Recommandation spécifie la représentation MHEG d'échange de scripts (SIR MHEG) pour les contenus des objets scripts, à savoir le codage des composants constituant les données des scripts de la classe script de MHEG.

Les moteurs MHEG sont des composants applicatifs ou systèmes qui manipulent, interprètent et présentent des objets MHEG. La présente Recommandation spécifie aussi la sémantique des scripts échangés. Cette sémantique est définie comme les contraintes minimales à prendre en compte dans le comportement des moteurs MHEG assurant l'interprétation des scripts échangés.

La présente Recommandation est applicable à toutes les applications échangeant des informations de natures multimédia et hypermédia.

2 Références normatives

La présente Recommandation se réfère à certaines dispositions des Recommandations UIT-T et textes suivants qui de ce fait en sont partie intégrante. Les versions indiquées étaient en vigueur au moment de la publication de la présente Recommandation. Toute Recommandation ou texte étant sujet à révision, les utilisateurs de la présente Recommandation sont invités à se reporter, si possible, aux versions les plus récentes des références normatives suivantes. La liste des Recommandations de l'UIT-T en vigueur est régulièrement publiée.

[1] Recommandation UIT-T X.680 (1994) | ISO/CEI 8824-1:1995, *Technologies de l'information – Notation de syntaxe abstraite numéro 1*: *spécification de la notation de base*.

[2] Recommandation UIT-T X.690 (1994) | ISO/CEI 8825-1:1995, *Technologies de l'information – Règles de codage de la notation de syntaxe abstraite numéro un: spécification des règles de codage de base, des règles de codage canoniques et des règles de codage distinctives*.

[3] Recommandations UIT-T X.290 à X.296 (1995), *Cadre général et méthodologie des tests de conformité OSI pour les Recommandations sur les protocoles pour les applications de l'UIT-T.*

ISO/CEI 9646 Parties 1 à 5, *Technologies de l'information – Interconnexion de systèmes ouverts – Cadre général et méthodologie des tests de conformité OSI.*

Partie 1: 1994, Concepts généraux.

Partie 2: 1994, Spécification des suites de tests abstraites.

Partie 3: 1992, Notation combinée erborescente et tabulaire (TTCN).

Partie 4: 1994, Réalisation des tests.

Partie 5: 1994, Spécifications pour laboratoires d'essais et clients pour le procédé d'évaluation de conformité.

[4] ISO/CEI 10646-1:1993, *Technologies de l'information – Jeu universel de caractères codés à plusieurs octets – Partie 1: Architecture et table multilingue.*

[5] Recommandation UIT-T T.171(1996), *Protocoles pour les services audiovisuels interactifs: représentation codée des objets multimédias et hypermédias.*

ISO/CEI 13522-1:1997, *Technologies de l'information – Codage de l'information multimédia et hypermédia – Partie 1: Représentation d'objet MHEG – Notation de base (ASN.1).*

[6] ISO/CEI 13522-4:1996, *Technologies de l'information – Représentation codée d'informations multimédia et hypermédia – Partie 4: Procédure d'enregistrement MHEG.*

[7] Recommandation UIT-T T.172 (1998), *MHEG-5 – Support pour applications interactives de niveau fondamental.*

ISO/CEI 13522-5:1997, *Technologies de l'information – Codage de l'information multimédia et hypermédia – Partie 5: Support pour applications interactives de niveau fondamental.*

[8] ISO/CEI 14750[1], *Technologies de l'information – Traitement distribué ouvert – Langage de définition d'interface. (Ancien DIS 14750-1.)*

[9] IEEE 754:1985, *IEEE standard for binary floating-point arithmetic*.

3 Définitions

3.1 Définitions

Pour les besoins de la présente Recommandation, les définitions données dans la Rec. UIT-T X.680 | ISO/CEI 8824-1 [1], Rec. UIT-T X.690 | ISO/CEI 8825-1 [2] ainsi que les définitions qui suivent y sont applicables.

3.1.1 interface programmatique d'application (API, *application programming interface*): frontière au-delà de laquelle une application utilise des fonctionnalités de langage de programmation pour invoquer des services logiciels. Ces fonctionnalités peuvent inclure des procédures ou opérations, des objets contenant des données partagées et de la résolution d'identificateurs.

3.1.2 attribut:

1) attribut MHEG (voir ISO/CEI 13522-1 [5]);

2) attribut IDL (q.v.).

3.1.3 moteur MHEG-3 conforme: moteur MHEG-3 dont l'implémentation est conforme au contenu de la présente Recommandation.

3.1.4 script MHEG-3 échangé conforme: script d'échange conforme au contenu de la présente Recommandation.

3.1.5 objet MHEG-3 conforme: objet script MHEG dont la représentation codée est conforme au contenu de la présente Recommandation.

---

[1] Actuellement à l'état de projet.

3.1.6 trame: enregistrement d'éléments dans la pile d'appel définissant un contexte d'exécution. Un tel enregistrement est empilé dans la pile d'appel à chaque appel de procédure pour y mémoriser le contexte courant d'exécution. Inversement, un tel enregistrement est dépilé de la pile d'appel en fin de procédure pour restaurer le contexte d'exécution d'avant l'appel.

3.1.7 hypermédia (adj.): le fait de gérer l'accès à de l'information monomédia et multimédia par le biais d'interactions auxquelles correspondent des liens explicites.

3.1.8 script échangé: la représentation codée de l'attribut "script data" d'un objet MHEG script.

3.1.9 langage de définition d'interface (IDL*, interface definition language*): notation formelle utilisée pour spécifier des types et des objets au travers de la définition de l'interface qu'ils fournissent selon l'ISO/CEI 14750-1 [8].

3.1.10 attribut IDL: association nommée et typée entre un objet et une valeur, déclarée comme partie de l'interface IDL et rendue visible des clients grâce à deux opérations: un extracteur (get) et un modificateur (set); en cas de lecture seule, on ne fournit que l'extracteur.

3.1.11 exception IDL: message apparaissant lorsqu'une une condition exceptionnelle se manifeste pendant le traitement d'une requête envers une opération IDL; celui-ci est défini comme un module IDL et peut avoir des membres qui sont retournés à l'appelant avec l'identificateur du message.

3.1.12 instance IDL: objet fournissant les opérations, les signatures et la sémantique spécifiées par une interface IDL; sa création et sa gestion dépendent de l'implémentation.

3.1.13 interface IDL: description IDL d'un ensemble d'opérations qu'un client peut requérir d'un objet IDL.

3.1.14 objet IDL: entité encapsulée et identifiable fournissant un ou plusieurs services pouvant être requis par un client.

3.1.15 opération IDL: service pouvant être requis qui est fourni par un objet IDL, celui-ci est défini au moyen d'une interface IDL par un nom, une signature définissant le type de ses paramètres et la valeur de retour ainsi que la liste des exceptions que son invocation peut faire apparaître.

3.1.16 mh-script: représentation interne, à l'intérieur d'un moteur MHEG, d'un objet MHEG script disponible.

3.1.17 action MHEG: opération appliquée aux objets MHEG formée de combinaisons séquentielles et parallèles d'actions élémentaires.

3.1.18 objet action MHEG: objet MHEG décrivant des actions MHEG.

3.1.19 application MHEG: application impliquant l'échange d'objets MHEG que ce soit en interne ou avec une autre application.

3.1.20 objet MHEG conforme: objet d'information dont la représentation codée est conforme aux dispositions d'une des parties de l'ISO/CEI 13522.

3.1.21 action élémentaire MHEG: l'une des opérations de base s'appliquant aux objets MHEG; elle correspond à une primitive de l'API MHEG.

3.1.22 moteur MHEG: processus ou ensemble de processus capable d'interpréter des objets MHEG.

3.1.23 entité MHEG: tout objet MHEG, *rt-object*, *content data*, *script data*, *socket*, *channel* ou autre construction définie dans l'ISO/CEI 13522.

3.1.24 lien MHEG: objet MHEG définissant des relations spatio-temporelles entre des objets MHEG en terme de conditions de déclenchement et d'actions.

3.1.25 objet MHEG: représentation codée d'une instance de classe d'objets MHEG.

3.1.26 classe MHEG script: classe MHEG définissant une structure pour échanger des données de script sous une forme codée spécifiée.

3.1.27 objet MHEG script: la représentation codée d'une instance de classe MHEG script.

3.1.28 API MHEG: l'API fournie par un moteur MHEG aux applications MHEG pour la manipulation d'objets MHEG.

3.1.29 MHEG-3 (adj.): s'applique aux entités conformes aux dispositions de la présente Recommandation.

3.1.30 application MHEG-3: application MHEG s'échangeant des scripts ou les échangeant avec d'autres applications. Ces scripts sont les composants *script data* d'objets MHEG scripts conformes à la représentation spécifiée par la présente Recommandation.

3.1.31 moteur MHEG-3: moteur MHEG traitant et interprétant les scripts de la SIR MHEG échangés.

3.1.32 profil MHEG-3: profil de la présente Recommandation.

3.1.33 SIR MHEG:

1) la représentation d'échange de script définie dans la présente Recommandation;

2) (adj.) s'applique à toute entité définie comme une partie de cette représentation d'échange de script (SIR, *script interchange représentation*).

3.1.34 pile d'appel de la SIR MHEG: pile associée à chaque *rt-script* en cours d'exécution par la machine virtuelle SIR MHEG, contenant une trame d'appel pour chaque invocation de fonction active.

3.1.35 code SIR MHEG: séquence codée d'instructions de la SIR MHEG.

3.1.36 constante (SIR MHEG): valeur statique, typée ou nommée déclarée à l'intérieur d'un script échangé, dont la valeur est accessible globalement et inchangée tout au long de l'exécution d'un script.

3.1.37 type construit (SIR MHEG): type décrit comme une combinaison d'autres types utilisant un des constructeurs suivants: séquence, chaîne, tableau, union, structure.

3.1.38 identificateur de données (SIR MHEG): entier identifiant de manière unique le nom d'un élément de donnée d'un script échangé (constante, variable globale, variable dynamique, variable locale).

3.1.39 exception (SIR MHEG): message déclenché durant l'invocation d'un service.

3.1.40 fonction (SIR MHEG): séquence nommée de code dont l'exécution peut être invoquée par un script échangé; ce peut être une routine, une fonction prédéfinie ou un service.

3.1.41 identificateur de fonction (SIR MHEG): entier identifiant de manière unique une fonction à l'intérieur d'un script échangé.

3.1.42 variable globale (SIR MHEG): variable de portée globale.

3.1.43 instruction (SIR MHEG): unité élémentaire de code d'un script SIR MHEG échangé; elle est formée d'un code opérateur suivi de zéro ou plusieurs opérandes.

3.1.44 unité d'exécution d'instruction (SIR MHEG): unité de traitement virtuel à l'intérieur d'un interpréteur de script SIR MHEG exécutant une instruction SIR MHEG.

3.1.45 script échangé (SIR MHEG): script échangé codé selon la SIR MHEG.

3.1.46 variable locale (SIR MHEG): variable dont la portée est locale à une routine.

3.1.47 message (SIR MHEG): message pouvant être reçu par l'interpréteur de script pendant l'exécution d'un script; il peut être prédéfini (exception de l'API MHEG, exception et opération de l'API MHEG-3, exception interne) ou déclaré à l'intérieur d'un script échangé (exception fournie par une interface externe).

3.1.48 identificateur de message (SIR MHEG): entier identifiant de manière unique un message à l'intérieur d'un script échangé.

3.1.49 file d'attente de message (SIR MHEG): file d'attente associée par la machine virtuelle SIR MHEG à chaque *rt-script* en cours d'exécution et contenant les messages à destination du *rt-script*.

3.1.50 référence à objet (SIR MHEG): valeur SIR MHEG, représentant une instance IDL, passée comme paramètre d'un appel externe pour requérir un service de cette instance.

3.1.51 opérande (SIR MHEG): paramètre d'instruction, il est codé à côté du code opérateur d'instruction.

3.1.52 bloc (SIR MHEG): ensemble des fonctions externes fournies par un module de l'environnement d'exécution, et qui sont accessibles par un *rt-script* et déclarées dans un script échangé. Il se compose de services et d'exceptions.

3.1.53 paramètre (SIR MHEG): données échangées avec un appel de fonction, un message ou une instruction.

3.1.54 pile de paramètres (SIR MHEG): pile associée par la machine virtuelle SIR MHEG à chaque *rt-script* en cours d'exécution et utilisée pour fournir des paramètres aux instructions et en récupérer les résultats en retour.

3.1.55 type prédéfini (SIR MHEG): type dont la description et l'identificateur étant prédéfinis dans la présente Recommandation ne nécessite pas d'être déclaré dans les scripts échangés; ce peut être soit un type primitif ou un type construit.

3.1.56 type primitif (SIR MHEG): type de base prédéfini par opposition au type construit.

3.1.57 routine (SIR MHEG): fonction déclarée dans un script échangé avec le code machine virtuelle en définissant la sémantique.

3.1.58 unité d'exécution de *rt-script* (SIR MHEG): unité de traitement virtuel, à l'intérieur d'un interpréteur de script SIR MHEG, exécutant du code de script.

3.1.59 interpréteur de script (SIR MHEG): la partie d'un moteur MHEG-3 manipulant et interprétant les scripts échangés.

3.1.60 service (SIR MHEG): à l'intérieur d'un script échangé, fonction externe dont l'implémentation est rendue accessible à un *rt-script* par l'environnement d'exécution de la plate-forme d'exécution.

3.1.61 variable (SIR MHEG): unité de mémoire typée et nommée de la machine virtuelle SIR MHEG dont la valeur peut changer à tout instant lorsque son domaine d'action est actif et dont on peut lire la valeur la plus récente.

3.1.62 machine virtuelle (SIR MHEG): description abstraite d'unités de mémoire et moteur d'exécution d'instructions d'un interpréteur de script SIR MHEG.

3.1.63 multimedia (adj.): le fait de manipuler plusieurs types de média de représentation.

3.1.64 application multimédia et hypermédia: application traitant la présentation d'informations multimédias à destination de l'utilisateur ainsi que la navigation interactive dans cette information.

3.1.65 application multimédia: application traitant la présentation de l'information multimédia à destination de l'utilisateur.

3.1.66 spécification de mappage de plate-forme: spécification de la manière dont les implémentations de moteur MHEG-3 doivent faire mapper les spécifications IDL aux composants de l'environnement d'exécution sur un type de plate forme donné.

3.1.67 file d'attente: collection d'éléments insérés et ôtés selon un ordre de type premier entré premier sorti.

3.1.68 *rt-script*: instance d'exécution (ou copie) d'un *mh-script*, créée par un moteur MHEG.

3.1.69 domaine de visibilité: contexte de référence pour une variable: si elle est globale, la variable peut être référencée par toute instruction de script; si elle est locale, la variable peut être référencée seulement dans le contexte local d'exécution.

3.1.70 langage de script: langage de programmation ayant pour objectif de permettre à des programmeurs non professionnels de réaliser rapidement et facilement des applications.

3.1.71 représentation de script pour les échanges (SIR, *script interchange representation*): représentation codée utilisée par une application pour échanger des scripts dans le but d'implémenter un comportement dynamique.

3.1.72 pile: collection d'éléments insérés (empilés) et ôtés (dépilés) selon un ordre de type dernier entré premier sorti.

3.2 Abréviations

La présente Recommandation utilise les abréviations suivantes:

API	interface programmatique d'application *(application programming interface)*
ASN.1	notation de syntaxe abstraite numéro 1 *(abstract syntax notation one)*
CEI	Commission électrotechnique internationale
CORBA	*Common Object Request Broker Architecture*
CS	pile d'appel *(call stack*)
CT	table de constantes *(constant table)*
DER	règles de codage distinctives *(distinguished encoding rules)*
DID	identificateur de données *(data identifier)*
DT	table de données *(data table)*
EBNF	formalisme de Backus-Naur étendu *(extended Backus-Naur form)*
ER	registre d'erreur *(error register)*
ETR	rapport technique ETSI *(ETSI technical report)*
FID	identificateur de fonction *(function identifier)*
FIFO	premier entré premier sorti *(first in first out)*
FP	pointeur de trame *(frame pointer)*
FR	registre de fonction *(function register)*
GT	table de définition de variables globales *(global variable definition table)*
HT	table de définition de filets *(handler definition table)*

IDL	langage de définition d'interface *(interface definition language)*
IP	pointeur d'instruction *(instruction pointer)*
IR	registre d'instruction *(instruction register)*
ISO	Organisation internationale de normalisation *(International organization for standardization)*
ITU-T	Union internationale des télécommunications – Secteur de la normalisation des télécommunications
JTC	Comité technique commun *(Joint technical committee)*
LIFO	dernier entré dernier sorti *(last-in first-out)*
LT	table de variables locales *(local variable table)*
MHEG	groupe d'experts pour le codage de l'information multimédia et hypermédia *(multimedia and hypermedia information coding experts group)*
MID	identificateur de message *(message identifier)*
MPEG/DSM-CC	groupe d'experts pour les images animées – commande et contrôle du support de stockage numérique *(moving picture experts group – digital storage media command and control)*
MQ	file d'attente de messages *(message queue)*
PID	identificateur de bloc *(package identifier)*
PS	pile de paramètres (*parameter stack)*
PT	table de définition de blocs (*package definition table)*
QP	pointeur de file d'attente (*queue pointer)*
rt	exécution (*run-time)*
RT	table de définition de routines (*routine definition table)*
SIR	représentation de script pour les échanges (*script interchange representation)*
SP	pointeur de pile (*stack pointer)*
ST	table de définition de services (*service definition table)*
TID	identificateur de type (*type identifier)*
TLV	valeur de longueur de type (*type-length-value)*
TT	table de définition de types (*type definition table)*
VT	table de variables (*variable table)*
XT	table de définition d'exceptions *(exception definition table)*

4 Généralités

Les Recommandations UIT-T de la série T.170 (et de l'ISO/CEI 13522) définissent la représentation codée des objets d'information multimédia/hypermédia (Objets MHEG) en vue de leur échange sous forme définitive entre des services et des applications, à l'aide de tout moyen d'échange incluant les réseaux locaux, les réseaux étendus de télécommunication et de télédiffusion, les supports de stockage, etc.

Les objets MHEG sont habituellement produits par des outils informatiques prenant comme forme source des applications multimédias réalisées avec des langages de script multimédias. Dans ce contexte, l'une des classes d'objets MHEG a pour objectif de compléter les autres classes MHEG en permettant d'exprimer les fonctionnalités habituellement supportées par les langages de script. Les objets scripts permettent d'exprimer des mécanismes de contrôle puissants et de décrire des relations plus complexes entre les objets MHEG que celles qui peuvent être exprimées par les seuls objets MHEG liens et actions. De plus, les objets scripts permettent d'exprimer l'accès et l'interaction avec des services externes fournis par l'environnement d'exécution.

D'autres Recommandations de la série T.170 définissent une représentation codée des objets scripts de manière ouverte de sorte que les objets scripts puissent encapsuler du code de script normalisé ou propriétaire. Les objets scripts encapsulent des scripts pouvant être codés dans tout format de codage enregistré dans l'ISO/CEI 13522-4 [6].

5 Conformité

La présente Recommandation définit des critères de conformité:

– sur les objets d'information, à savoir les objets scripts MHEG;

– sur les implémentations, à savoir les implémentations de moteurs MHEG.

5.1 Conformité d'un objet d'information

Un objet script MHEG-3 conforme devra satisfaire à tous les critères suivants:

1) sa représentation codée devra être conforme aux dispositions des Recommandations UIT-T de la série T.170 (et des parties de l'ISO/CEI 13522);

2) sa représentation codée devra contenir un script MHEG-3 échangé conforme.

La conformité d'un objet d'information est évaluée sur les objets d'information échangés dans le but d'être exécutés sur un terminal.

5.1.1 Profils

La présente Recommandation ne définit pas de profils.

NOTE 1 – Cependant des profils MHEG-3 peuvent être définis par d'autres normes ou par d'autres parties de l'ISO/CEI 13522. En accord avec le schéma de définition de profil, les profils MHEG-3 normalisés devraient au moins être aussi contraignants; les objets d'information se déclarant conformes à de tels profils devraient au moins être conforme à la présente Recommandation.

Un profil MHEG-3 doit définir ce qui suit:

– un profil de la machine virtuelle SIR MHEG définie dans la présente Recommandation;

– un profil IDL et son mappage avec la SIR MHEG pour exprimer l'interface entre les scripts et les environnements externes;

– une API pour la manipulation des objets MHEG définis dans les Recommandations UIT-T de la série T.170 (et des parties de l'ISO/CEI 13522) avec un mappage entre cette interface et la SIR MHEG.

NOTE 2 – En accord avec les normes ISO, les profils MHEG-3 devraient assurer une compatibilité amont du codage ASN.1 de sorte que les scripts échangés soient conformes au profil MHEG-3 lui aussi conforme à la présente Recommandation.

5.1.2 Codage

Un script échangé MHEG-3 conforme devra être codé selon les règles de codage définies dans l'Annexe B.

5.1.3 Syntaxe

Un script échangé MHEG-3 conforme devra être conforme à la syntaxe ASN.1 définie dans l'Annexe A.

5.1.4 Sémantique

Un script échangé MHEG-3 conforme devra inclure des déclarations valides du point de vue sémantique et des séquences d'instructions selon les définitions des paragraphes 12 et 13.

5.2 Conformité d'implémentation

Toute implémentation de la présente Recommandation est un moteur MHEG-3.

Un moteur MHEG-3 conforme devra supporter l'interprétation d'objets scripts MHEG-3 conformes.

La présente Recommandation définit la sémantique des scripts MHEG-3 échangés. Cela n'implique pas des besoins de conformité sur les objets d'information mais plutôt sur le comportement des moteurs MHEG-3.

NOTE 1– Bien qu'un script MHEG-3 conforme puisse ne pas reproduire la sémantique impliquée par son réalisateur, la manière dont les moteurs conformes se comportent dans l'interprétation de ce script est prévisible.

NOTE 2– La présente Recommandation ne considère pas la conformité d'un système, d'un moteur ou d'un processus à partir du moment où celle-ci n'a pas de lien avec l'interprétation des scripts échangés.

5.2.1 Requêtes de conformité

Un moteur MHEG-3 conforme devra satisfaire à tous les critères suivants:

1) il devra analyser la syntaxe et interpréter les scripts MHEG-3 échangés selon le comportement de machine virtuelle défini dans la présente Recommandation (voir paragraphe 9);
2) il devra supporter la communication avec l'environnement d'exécution et les objets MHEG selon le schéma de mappage IDL défini dans la présente Recommandation (voir paragraphes 10, 11 et 14);
3) il devra fournir l'API MHEG-3 définie dans la présente Recommandation (voir paragraphe 15 et Annexe F);
4) afin de permettre la manipulation d'objets MHEG par les scripts échangés, il devra supporter une API MHEG et son mappage selon le schéma défini dans la présente Recommandation (voir Annexe E);
5) afin de permettre la communication avec l'environnement d'exécution, il devra supporter une spécification de mappage de plate-forme selon le schéma défini dans la présente Recommandation (voir Annexe D);
6) il peut fournir des fonctions et des fonctionnalités non requises par la présente Recommandation ou par la spécification de mappage de plate-forme. Une telle extension non normalisée devra être identifiée comme telle dans la documentation système.

5.2.2 Documentation de conformité

Un document de conformité contenant l'information suivante devra être disponible pour toute implémentation se réclamant conforme à la présente Recommandation. Le document de conformité devra satisfaire à tous les critères suivants:

1) il devra lister tous les attributs obligatoires requis dans la présente Recommandation, avec référence aux paragraphes et sous-paragraphes adéquats;

2) il devra soit inclure la spécification de mappage de plate-forme à laquelle l'implémentation est conforme ou référencer sans ambiguïté une spécification de mappage de plate-forme enregistrée;

3) il devra contenir un état indiquant les noms complets, nombres et dates des normes qui s'y appliquent.;

4) il devra faire état de ceux des attributs optionnels définis dans la présente Recommandation et dans la spécification de mappage de plate-forme qui sont supportés par l'implémentation;

5) il devra décrire le comportement durant l'implémentation pour tous les attributs de l'implémentation définis dans la présente Recommandation et dans la spécification de mappage de plate-forme. On répondra à ce besoin en listant ces attributs et en fournissant soit une référence spécifique à la documentation système, soit une syntaxe et une sémantique complètes de ces attributs. Le document de conformité peut spécifier le comportement durant l'implémentation pour les attributs dans lesquels la présente Recommandation ou la spécification de mappage de plate-forme établit que les implémentations peuvent varier ou dans lesquels des attributs sont identifiés comme indéfinis ou non spécifiés.

Aucune autre spécification que celles spécifiées dans la présente Recommandation et dans la spécification de mappage de plate-forme ne devra être présente dans le document de conformité.

5.3 Conformité d'application

Une application de la présente Recommandation (appelée application MHEG-3) est une application MHEG s'échangeant des scripts et les échangeant avec d'autres applications en tant que composants *script data* des objets MHEG scripts selon la représentation codée spécifiée dans la présente Recommandation.

5.4 Méthodes des test

Toute mesure de conformité à la présente Recommandation devra être effectuée en utilisant des méthodes de test conformes aux Recommandations UIT-T de la série X.290 (et de l'ISO/CEI 9646) [3].

6 Aperçu général

La présente Recommandation étend les dispositions des autres Recommandations UIT-T de la série T.170 (et des autres parties de l'ISO/CEI 13522) pour faire en sorte que les objets et les applications MHEG puissent supporter des fonctionnalités de langages de script multimédias de manière normalisée. Si l'on considère les fonctionnalités supportées par les autres Recommandations UIT-T de la série T.170 (et des parties de l'ISO/CEI 13522), ces extensions sont décomposées en deux thèmes principaux:

– opérations de traitement de données (voir sous-paragraphe 6.2);

– accès aux données et fonctions externes (voir sous-paragraphe 6.3).

Pour étayer ces deux thèmes, la présente Recommandation spécifie:

– des dispositions complètes et détaillées pour le codage des scripts échangés;
– le comportement attendu d'un interpréteur de script.

6.1 Méthodologie de description

La description des dispositions de la présente Recommandation suit une méthodologie comportant quatre niveaux de description:

– niveau a): description textuelle informelle;
– niveau b): description détaillée de la sémantique;
– niveau c): description formelle de la syntaxe;
– niveau d): description formelle du codage.

Ces niveaux sont utilisées dans les paragraphes de la manière suivante:

– niveau a): paragraphes 8 à 11;
– niveau b): paragraphes 12 à 15;
– niveau c): Annexes A, E, F, G;
– niveau d): Annexes B, C.

NOTE – Les Appendices informatifs I et II utilisent aussi le niveau c) de description.

6.2 Opérations de traitement de données

Pour traiter les opérations de traitement de données, la SIR MHEG définit la structure des scripts échangés. Celle-ci est composée de déclarations de données et de déclarations de fonctions, ces dernières encapsulant des séquences d'instructions.

Le paragraphe 8 définit les éléments du code de la machine virtuelle SIR MHEG.

Le paragraphe 9 spécifie la machine virtuelle SIR MHEG, à savoir un modèle définissant comment les interpréteurs de script SIR MHEG devront effectuer l'interprétation du code d'un script SIR MHEG. Cette machine virtuelle est utilisée ensuite pour décrire la sémantique des instructions SIR MHEG. Le paragraphe 9 fait état des besoins sur les fonctionnalités que les interpréteurs de script devront fournir; cependant, elle ne spécifie pas comment implémenter ces fonctionnalités.

Le paragraphe 12 définit les déclarations des scripts SIR MHEG échangés. Elle spécifie leur structure (à savoir la manière dont elles devront être représentées) ainsi que leur sémantique (à savoir la manière dont elles devront être interprétées par des interpréteurs de script SIR MHEG). La sémantique est spécifiée en utilisant le formalisme de machine virtuelle introduit dans le paragraphe 9.

Le paragraphe 13 définit les instructions SIR MHEG. Elle en spécifie la structure (à savoir la manière dont elles devront être représentées) ainsi que la sémantique (à savoir la manière dont elles devront être interprétés par les interpréteurs de script SIR MHEG). La sémantique est spécifiée en utilisant le formalisme de machine virtuelle introduit dans le paragraphe 9.

L'Annexe A définit formellement la syntaxe détaillée des scripts échangés en utilisant la notation ASN.1.

L'Annexe B définit formellement le codage des scripts échangés.

L'Annexe C dresse une liste des éléments prédéfinis de la SIR MHEG et en définit le codage.

L'Annexe G définit formellement comment la présente Recommandation s'instancie dans les Rec. UIT-T T.171 (et ISO/CEI 13522-1), T.172 (et ISO/CEI 13522-5), à savoir les objets MHEG de ces parties auxquels s'applique la SIR MHEG. ainsi que la manière de l'appliquer.

6.3 Accès aux données et fonctions externes

Le traitement de l'accès aux données et fonctions externes dans la SIR MHEG s'effectue en utilisant IDL pour décrire les interfaces de manière abstraite et indépendante du langage. Ceci permet d'unifier la manière dont les données et fonctions externes sont vues par les interpréteurs de script.

Dans le contexte de la SIR MHEG, on utilise IDL pour séparer clairement la manière (spécifique à la SIR MHEG) dont l'utilisation des données et fonctions externes est exprimée dans les scripts échangés de la manière (au moins dépendante de la plate-forme et peut-être dépendante de l'application) dont ces données et fonctions sont fournies par l'environnement externe. La SIR MHEG définit l'utilisation des interfaces tandis que la responsabilité de la définition de leur fourniture est laissée à l'application.

Afin de permettre aux interpréteurs de script de manipuler des informations MHEG et d'échanger avec elles des informations, les moteurs MHEG-3 fournissent aux interpréteurs de script l'accès aux entités MHEG (données) et la possibilité d'invoquer les actions MHEG (fonctions) grâce à une API MHEG définie en IDL. Les types et les actions MHEG sont prédéfinis dans la SIR MHEG pour assurer un codage compact et une interprétation efficace lors de la manipulation d'objets MHEG.

Afin de permettre aux interpréteurs de scripts de coopérer avec l'environnement d'exécution, celui-ci fournit un accès à ses données et ses fonctions selon une spécification de mappage de plate-forme IDL. Cette spécification décrit comment doit-on fournir les opérations IDL sur une plate-forme particulière pour que les moteurs MHEG-3 soient capables de les utiliser comme des services externes.

NOTE – Les blocs peuvent être fournis sous la forme de bibliothèques, pilotes de dispositifs, composants de systèmes d'exploitation, processus, services de télécommunications, etc.

Le paragraphe 7 décrit des hypothèses sur la structure des moteurs MHEG-3 et leurs relations avec l'environnement.

Le paragraphe 10 décrit les mécanismes généraux utilisés qui sont fournis par l'environnement d'exploitation pour accéder aux données et fonctions externes.

Le paragraphe 11 décrit les mécanismes généraux utilisés pour manipuler des objets MHEG.

Le paragraphe 14 spécifie le mappage IDL pour la SIR MHEG, à savoir les mécanismes utilisés par la représentation SIR MHEG pour décrire les blocs et invoquer les opérations IDL.

Le paragraphe 15 spécifie la structure et la sémantique de l'API MHEG-3, à savoir l'ensemble des opérations pouvant être utilisées pour manipuler des scripts.

L'Annexe D spécifie le formulaire IDL de spécification de mappage de plate-forme, à savoir le document à remplir et enregistrer pour chaque type de plate-forme décrivant les dispositions spécifiques à la plate-forme que doivent remplir les services fournis par l'environnement d'exploitation de cette plate-forme et auxquelles les moteurs MHEG-3 doivent se conformer pour pouvoir coopérer avec les services fournis par l'environnement d'exploitation de cette plate-forme et ainsi, interpréter les scripts qui appellent ces services.

L'Annexe E spécifie le cadre à utiliser pour la définition d'une API MHEG utilisant IDL et la procédure à suivre pour la faire mapper à la SIR MHEG.

L'Annexe F définit la syntaxe détaillée de l'API MHEG-3 en notation IDL.

7 Relations entre MHEG et MHEG-3

Le présent paragraphe introduit des hypothèses générales sur les moteurs MHEG-3; celles-ci sont ensuite utilisées pour décrire la performance d'un interpréteur de script et ses relations avec l'environnement externe.

Les moteurs MHEG-3 devront fournir les fonctionnalités décrites ci-après afin de présenter le comportement attendu pour tout ce qui concerne l'interprétation des scripts échangés.

Cependant, les moteurs MHEG-3 ne sont pas obligés d'implémenter les fonctionnalités décrites.

NOTE – Par exemple, les composants fonctionnels du moteur MHEG-3 décrits ci-après n'ont pas à correspondre à des composants réels (logiciels) d'implémentations de moteurs MHEG-3.

7.1 Entités MHEG

Les entités MHEG-3 manipulent des entités MHEG: objets MHEG, *mh-objets*, *rt-objects,* objets MHEG échangés, supports, canaux.

NOTE – les entités MHEG sont décrites plus en détail dans l'Appendice III.

7.2 Entités fonctionnelles

Les moteurs MHEG-3 peuvent être considérés comme un assemblage des composants fonctionnels suivants:

– analyseur syntaxique d'objet MHEG: analyse la syntaxe des objets MHEG échangés et les transforme en *mh-objects* sous le contrôle du gestionnaire de *mh-objets*;

– gestionnaire de *mh-object*: contrôle leur cycle de vie et permet l'accès à tous les *mh-objects*;

– gestionnaire de *rt-object*: contrôle leur cycle de vie et permet l'accès à tous les *rt-objects*

– résolveur de référence: transforme une référence MHEG en un identificateur ou un gestionnaire utilisable;

– gestionnaire de lien: surveille les liens actifs et déclenche les actions correspondantes lorsque leurs conditions sont à vrai;

– interpréteur d'action: interprète les actions MHEG élémentaires;

– interpréteur de script: analyse la syntaxe des scripts SIR MHEG échangés et en interprète les *rt-scripts* , fournit l'accès à l'environnement d'exécution;

– agent de présentation: assure l'interface avec l'environnement de présentation; commande la présentation des *rt-contents* ; reçoit les sélections et modifications de l'utilisateur;

– agent d'accès: assure l'interface avec l'environnement de communication; fournit l'accès aux objets MHEG échangés et aux données des contenus;

7.3 Interpréteur de représentation d'échange de script (SIR MHEG)

Dans un moteur MHEG-3, l'interpréteur de script aura pour tâche la responsabilité de ce qui suit:

– analyser la syntaxe des scripts échangés (fournis par l'analyseur syntaxique d'objets MHEG)

– préparer les structures de données adéquates pour l'exécution ultérieure des *rt-scripts;*

– exécuter le code du script;

– réaliser le *default effect* des actions MHEG à destination des *mh-scripts* et des *rt-scripts*;

– invoquer les filets appropriés (dans le programme du script) pour ces actions MHEG;

– transmettre les actions MHEG élémentaires invoquées par le programme du script à l'interpréteur d'action;

– Gérer l'échange avec l'environnement d'exécution (localiser et charger les blocs, invoquer les services, recevoir les messages, passer les données) en utilisant les mécanismes appropriés de communication spécifiques à la plate-forme;

8 Composants de la représentation de script pour les échanges MHEG (SIR MHEG)

Le présent paragraphe décrit les principaux éléments de la SIR MHEG ainsi que la manière dont les scripts échangés les utiliseront.

Les entités déclarées et manipulées par les scripts SIR MHEG échangés sont des:

– types de données;

– données;

– fonctions;

– messages.

Ces concepts sont définis dans les sous-paragraphes suivants; cependant, la structure détaillée de leurs déclarations est spécifiée dans le paragraphe 12.

8.1 Types de données

Les types de données sont utilisés pour décrire la structure:

– des données propriétaires de script (constantes et variables);

– des paramètres et des valeurs de retour des routines des scripts;

– des paramètres et des valeurs de retour des fonctions externes;

– des paramètres et des messages manipulés par les scripts.

Comme il y a besoin d'adapter les scripts eux-mêmes à la signature des fonctions pouvant être fournies par l'environnement externe, la SIR MHEG définit toute une gamme de types correspondant aux types de données IDL.

Le codage des définitions des types de données dans un script échangé est défini dans l'Annexe A. La présente Recommandation n'impose pas de condition sur la manière dont les moteurs MHEG-3 représentent ces types de données.

La SIR MHEG utilise deux sortes de types de données:

– types prédéfinis (voir 8.1.1);

– type déclarés (voir 8.1.2).

Tous les types peuvent être référencés de manière unique et non ambiguë grâce à leur identificateur de type.

8.1.1 Types prédéfinis

Les types prédéfinis peuvent être soit des types primitifs ou construits:

Les types prédéfinis ont des identificateurs de types prédéfinis et ainsi n'ont pas besoin d'être déclarés dans les scripts échangés. La liste des types prédéfinis et leurs identificateurs est donnée en Annexe C.

8.1.1.1 Types primitifs

Les type primitifs correspondent aux types primitifs IDL. Voici la liste des types primitifs de la SIR MHEG.

– **void**;

– **octet**;
– **short**;
– **long**;
– **unsigned short**;
– **unsigned long**;
– **float**;
– **double**;
– **boolean**;
– **character**;
– **data identifier**;
– **object reference**.

Afin d'en faciliter la référence, les types primitifs ont tous un code correspondant à une lettre comme indiqué dans le Tableau 1.

Tableau 1/T.173 – Codes des lettres des types primitifs

Type	Code lettre
octet	O
short	S
long	L
unsigned short	W (comme word en anglais)
unsigned long	U
float	F
double	D
boolean	B
character	C
data identifier	I (comme identificateur)
object reference	R (comme référence)

8.1.1.1.1 Type **void**

Le type **void** devra être utilisé seulement pour exprimer le type de la valeur de retour d'une fonction. Les fonctions dont le type de la valeur de retour est **void** ne retournent aucune donnée. Un script échangé n'aura jamais de constantes et de variables de type **void**. Le type **void** ne sera pas utilisé lors de la définition de types construits.

8.1.1.1.2 Type **octet**

Les données dont le type est **octet** prendront une valeur numérique comprise dans l'intervalle [0 .. 255]. Les variables **octet** sans valeur initiale explicite seront initialisées à 0.

8.1.1.1.3 Type **short**

Les données dont le type est **short** pourront avoir une valeur entière signée dans l'intervalle [–32 768 .. 32 767]. Les variables **short** sans valeur initiale explicite seront initialisées à 0.

8.1.1.1.4 Type **long**

Les données dont le type est **long** pourront avoir une valeur entière signée dans l'intervalle [−2147483648 .. 2147483647]. Les variables **long** sans valeur initiale explicite seront initialisées à 0.

8.1.1.1.5 Type **unsigned short**

Les données dont le type est **unsigned short** pourront avoir une valeur entière signée dans l'intervalle [0 .. 65535]. Les variables **unsigned short** sans valeur initiale explicite seront initialisées à 0.

8.1.1.1.6 Type **unsigned long**

Les données dont le type est **unsigned long** pourront avoir une valeur entière signée dans l'intervalle [0 .. 4294967295]. Les variables **unsigned long** sans valeur initiale explicite seront initialisées à 0.

8.1.1.1.7 Type **float**

Les données dont le type est **float** pourront avoir une valeur de virgule flottante simple précision dans l'intervalle spécifié par IEEE 754 [9]. Les variables **float** sans valeur initiale explicite seront initialisées à 0.

8.1.1.1.8 Type **double**

Les données dont le type est **double** pourront avoir une valeur de virgule flottante double précision dans l'intervalle spécifié par IEEE 754 [9]. Les variables **double** sans valeur initiale explicite seront initialisées à 0.

8.1.1.1.9 Type **boolean**

Les données dont le type est **boolean** pourront avoir une valeur *true* (vrai) ou *false* (faux). Les variables **boolean** sans valeur initiale explicite seront initialisées à *false*.

8.1.1.1.10 Type **character**

Les données dont le type est **character** auront une valeur de caractère comprise dans le jeu de caractères **BMPString** défini par le Basic Multilingual Plane de l'ISO/CEI 10646-1 [4]. Les variables **character** sans valeur initiale explicite auront une valeur initiale indéfinie.

Les moteurs MHEG-3 peuvent retenir un ensemble restreint de caractères, fondé sur les jeux normalisés de l'Annexe A de l'ISO/CEI 10646-1 [4]. Dans ce cas, les sous-ensembles adoptés et leur niveau d'implémentation devront être documentés dans le document de conformité.

8.1.1.1.11 Type **data identifier (identificateur de donnée)**

Les données dont le type est **data identifier** pourront prendre une valeur entière non signée comprise dans l'intervalle [0 .. 65535]. Cette valeur est utilisée pour identifier soit une constante, une variable globale, une variable dynamique, une variable locale ou un paramètre de routine du script selon la définition donnée au 8.6.2. Une constante ne pourra pas être de type **data identifier**. Les variables **data identifier** sans valeur initiale explicite auront une valeur initiale indéfinie.

8.1.1.1.12 Type **object reference (références à objet)**

Les données dont le type est **object reference** prendront la valeur d'un filet (service) référençant un objet IDL s'appuyant sur des fonctions ou des services prédéfinis. Le codage des références à objet est défini par la spécification de mappage de la plate-forme. Il n'y aura pas de constante de type **object reference**. Le type **object reference** ne sera pas utilisé pour définir des types construits. Les variables **object reference** sans valeur initiale explicite prendront une valeur initiale indéfinie.

Les références à objet sont utilisées en tant que premier paramètre implicite de tous les appels externes pour spécifier l'objet sur lequel s'applique l'appel. Les valeurs des références à objet seront fournies par l'environnement externe comme valeur d'entrée ou paramètre de retour d'une instruction d'**appel externe (XCALL)**. L'instruction **get object reference** (**GETOR**) est utilisée pour extraire une première référence à objet de l'objet racine d'un bloc donné. La référence à objet **null** est utilisée pour faire référence à l'objet d'origine de l'API MHEG-3 (instance du **ScriptInterpreter**).

8.1.1.2 Types construits prédéfinis

Afin de permettre aux scripts d'exprimer plus facilement des manipulations de données MHEG, les types de données de l'API MHEG sont prédéfinis.

Quoique non définis dans les scripts échangés, les types construits prédéfinis, en tant que types construits prédéclarés, peuvent être exprimés à l'aide de constructeurs de types et de constructeurs d'identificateurs, comme ceux qui sont décrits au 8.1.2. Seuls les identificateurs de types prédéfinis seront utilisés pour exprimer la structure de types construits prédéfinis.

8.1.2 Types construits déclarés

Les types construits seront définis à l'aide d'un constructeur et d'un ou plusieurs identificateurs de type pour identifier soit un type déclaré ou un type prédéfini.

Les constructeurs d'un type construit pourront être l'un des suivants:

– **sequence** (voir 8.1.2.1);

– **string** (voir 8.1.2.2);

– **array** (voir 8.1.2.3);

– **structure** (voir 8.1.2.4);

– **union** (voir 8.1.2.5).

Les types déclarés sont définis à l'intérieur des types échangés.

Les types SIR MHEG ne pourront pas être redéfinis dans un script échangé. La structure d'un type déclaré ne devra pas correspondre à celle d'un type prédéfini ou à celle d'un autre type déclaré.

Il ne pourra y avoir plus de 16 384 types déclarés dans un script échangé.

8.1.2.1 Types **sequence** (séquence)

Les types **sequence** seront définis par:

– leur taille (optionnel);

– leur type d'élément.

La taille sera une valeur entière non signée. Elle représente le nombre maximal d'éléments de la séquence. Si la définition du type ne spécifie pas de taille, le nombre d'éléments peut avoir n'importe quelle taille jusqu'au maximum autorisé. Les types **sequence** ayant une taille explicite sont appelés des types **sequence** liés.

La taille maximale d'un type **sequence** est de 65 535 éléments.

Le type d'élément peut être tout type primitif, construit ou prédéfini sauf **void** et **object reference**. On référencera le type d'élément à l'aide de son identificateur de type. Les définitions de type **sequence** ne devront pas donner lieu à des récursions infinies.

NOTE – Il en résulte que l'identificateur de type **sequence** ne peut être niché à l'intérieur d'une définition de type que sous un constructeur **union**.

Les données dont le type est un type **sequence** défini prendront comme valeur une liste ordonnée de zéros ou plusieurs valeurs du type d'élément.

Les variables de type **sequence** n'ayant pas de valeur initiale explicite auront pour valeur initiale une liste nulle (séquence de zéro élément).

8.1.2.2 Type **string** (chaîne)

Les types **string** ont une sémantique équivalente à celle des types **sequence** dont l'élément est de type **character**.

NOTE – Pour en optimiser la manipulation, les valeurs de chaînes peuvent être implémentées d'une manière différente des séquences de caractères. Ainsi, les types de chaînes et de séquences de caractères restent distinctes, quoique sémantiquement équivalentes.

Les types **string** seront définis par leur taille (optionnel).

La taille sera une valeur entière non signée. Elle représente le nombre maximal d'éléments de la chaîne. Si la définition du type ne spécifie pas de taille, le nombre d'éléments peut avoir n'importe quelle taille jusqu'au maximum autorisé. Les types **string** ayant une taille explicite sont appelés des types **string** liés.

La taille maximale d'un type **string** est de 65 535 caractères.

Les données dont le type est un type **string** défini prendront comme valeur une liste ordonnée de zéros ou plusieurs caractères.

Les variables de type **string** n'ayant pas de valeur initiale auront pour valeur initiale une liste nulle (séquence de zéro caractère).

8.1.2.3 Type **array** (tableau)

Les types **array** seront définis par:

– leur taille;

– leur type d'élément.

La taille sera une valeur entière non signée. Elle représente le nombre exact d'éléments du tableau.

Le type d'élément peut être tout type primitif, construit ou prédéfini sauf **void** et **object reference**. On référencera le type d'élément en utilisant son identificateur. Les définitions de type **array** ne devront pas donner lieu à des récursions infinies.

NOTE – Il en résulte que l'identificateur de type **array** ne peut être niché à l'intérieur d'une définition de type que sous un constructeur **union**.

Les données dont le type est un type **array** prendront comme valeur une liste ordonnée de valeurs du type d'élément, la longueur de cette liste étant donnée par la taille du tableau.

Les variables d'un type **array** n'ayant pas de valeur initiale auront pour valeur initiale une liste d'éléments dont la valeur initiale est définie par le type de l'élément.

8.1.2.4 Type **structure** (structure)

Les types **structure** seront définis par une liste ordonnée allant de 1 à 256 types d'éléments.

Le type d'élément peut être tout type primitif, construit ou prédéfini sauf **void** et **object reference**. On référencera le type d'élément en utilisant ses identificateurs de types. Les définitions de type **structure** ne devront pas donner lieu à des récursions infinies.

NOTE – Il en résulte que l'identificateur de type **structure** ne peut être niché à l'intérieur d'une définition de type que sous un constructeur **union**.

Les données dont le type est un type **structure** prendront comme valeur une liste ordonnée de valeurs du type d'élément correspondant à leur rang dans la définition du type.

Les variables d'un type **structure** n'ayant pas de valeur initiale auront pour valeur initiale une liste d'éléments dont la valeur initiale est définie par le type de l'élément.

8.1.2.5 Types union (union)

Les types **union** seront définis par une liste ordonnée de types d'éléments.

Il n'y aura pas plus de 256 choix (types d'éléments) dans un type **union**.

Les types d'élément peuvent être tout type primitif, construit ou prédéfini sauf **void** et **object reference**. On référencera le type d'élément en utilisant ses identificateurs de types.

Les données dont le type est un type **union** prendront comme valeur:

– un entier représentant l'index (à partir de 0) dans la liste de choix;

– la valeur du type d'élément dont le rang dans la définition du type est l'index immédiatement supérieur.

Les variables d'un type **union** n'ayant pas de valeur initiale auront une valeur indéfinie.

8.2 Données

La SIR MHEG définit trois types de données:

– valeurs immédiates (voir 8.2.1);

– constantes (voir 8.2.2);

– variables (voir 8.2.3).

Toutes les données utilisés par un script échangé sont d'un type définitif qu'il soit prédéfini ou déclaré.

Deux valeurs de données seront égales si et seulement si:

– elles sont du même type, à savoir elles ont le même identificateur de type;

– si elles sont d'un type primitif alors elles sont identiques;

– si elles sont d'un type **structure**, **sequence** ou **array** alors chaque élément d'une liste est égal à l'élément de même rang dans l'autre liste;

– si elles sont d'un type **union** alors leurs étiquettes sont identiques et leurs valeurs sont égales deux à deux.

Il en résulte que

– les valeurs d'un type **string** ne devront pas être comparées avec celles d'une **sequence** de type **character** ;

– les valeurs d'un type **sequence** lié ne devront pas être comparées avec celles d'un autre type **sequence** lié ou avec celles d'un type **sequence** non lié, dans la mesure où elles ont des types d'identificateurs différents;

– les valeurs d'un type **string** lié ne devront pas être comparées avec celles d'un autre type **string** lié ou avec celles d'un type **string** non lié, dans la mesure où elles ont des types d'identificateurs différents;

Toutes les variables et les constantes sont référencées de manière unique et non ambiguë par leur identificateur de donnée.

8.2.1 Valeurs immédiates

Les valeurs immédiates sont des données qui ne sont pas déclarées avec les scripts échangés, et ainsi ne peuvent qu'être utilisées " immédiatement ", c'est-à-dire tel quel. Une valeur immédiate peut se trouver dans un script échangé:

– comme une valeur constante;

– comme la valeur initiale d'une variable;

– comme opérande de l'instruction **push immediate (PUSHI)**.

En outre, des valeurs immédiates sont utilisées tout au long de l'exécution d'un script à travers la pile de paramètres comme paramètres d'instructions ou de fonctions.

En dehors de toute restriction provenant du contexte, les valeurs immédiates peuvent être de tout type excepté le type void.

Le codage des valeurs des données dans un script échangé est défini par l'Annexe A. La présente Recommandation n'impose pas de directives sur la manière dont les moteurs MHEG-3 représentent les valeurs des données pour un type particulier.

8.2.2 Constantes

Les constantes devront être déclarées dans un script échangé et définies par:

– un type de donnée;

– une valeur de ce type de donnée.

Les constantes peuvent être de n'importe quel type excepté

– **object reference**;

– **data identifier**;

– **void**.

Les constantes ont une portée globale et peuvent être référencées à l'aide de leur identificateur de donnée d'un bout à l'autre du script échangé.

Il ne sera pas possible de déclarer plus de 4096 constantes dans un script échangé.

8.2.3 Variables

Les variables devront être déclarées dans un script échangé et définies par:

– un type de données;

– en option, une valeur pour ce type de donnée, à prendre comme valeur initiale.

Les variables peuvent être de n'importe quel type excepté **void.**

Les variables sont référencées en utilisant leur identificateur de donnée. Une référence à une variable peut être utilisée avec l'une des sémantiques suivantes:

– sémantique " droite ": comme si on fournissait la valeur de la variable à la place;

– sémantique " gauche ": indique qu'une valeur de donnée doit être affectée à cette variable.

Dans le dernier cas, la valeur assignée à la variable peut être soit une valeur immédiate (incluant une valeur calculée), la valeur d'une constante ou la valeur d'une variable (incluant la valeur à venir d'un paramètre de sortie de fonction).

Trois catégories de variables sont définies dans la SIR MHEG:

– variables globales (voir 8.2.3.1);

– variables locales (voir 8.2.3.2);

– variables dynamiques (voir 8.2.3.3).

8.2.3.1 Variables globales

Les variables globales ont une portée globale couvrant le script échangé tout entier. On peut les référencer en utilisant leur identificateur de donnée à partir de n'importe quelle routine ou variable. Une nouvelle valeur peut leur être assignée à tout moment de l'exécution d'un *rt-script*.

Il ne sera pas possible de déclarer plus de 28 672 variables dans un script échangé.

8.2.3.2 Variables locales

Les variables locales ont une portée lexicale restreinte à l'exécution du code de leur routine d'appartenance. On peut les référencer en utilisant leur identificateur de donnée à l'intérieur du code de cette routine.

Il existe deux types de variables locales:

– les variables locales déclarées à l'intérieur d'une routine comme faisant partie de la déclaration de variable locale;

– les paramètres effectifs de la routine, qu'ils soient passés par valeur ou par référence, déclarés à l'intérieur de la déclaration de routine comme faisant partie de la signature de cette routine.

Il ne sera pas possible de déclarer plus de 256 variables locales dans chaque routine d'un script échangé.

8.2.3.3 Variables dynamiques

Les variables dynamiques ont une portée dynamique pouvant s'étendre à partir de leur moment de création en utilisant l'instruction **allocate (ALLOC)** jusqu'au moment où elles seront libérées en utilisant l'instruction **free (FREE)**. L'interpréteur de script leur donne un identificateur de donnée lors de leur création. On peut les référencer à tout moment de l'exécution d'un script grâce à leur identificateur de donnée. Cependant, l'identificateur de donnée d'une variable dynamique étant seulement connu à l'exécution, celui-ci ne peut être utilisé que comme paramètre de pile ou valeur de variable et non comme opérande d'instruction.

Il ne sera pas possible d'utiliser plus de 32 512 variables dynamiques à un moment donné pendant l'exécution d'un *rt-script*.

8.3 Fonctions

La SIR MHEG définit trois types de fonctions:

– routines (voir 8.3.1);

– services (voir 8.3.2);

– fonctions prédéfinies (voir 8.3.3).

Toutes les fonctions devront avoir une signature (ou prototype) composé:

– d'un type de valeur de retour;

– d'une liste de paramètres formels définis par un type et un mode de passage.

L'utilisation d'un identificateur de fonction permet de les référencer de manière unique et non ambiguë.

Les fonctions seront soit **synchronous** ou **asynchronous**. En cas d'appel de fonction synchrone, l'appelant attend la fin de l'exécution de la fonction et peut ensuite récupérer son résultat. En cas

d'appel de fonction asynchrone, l'appelant attend seulement un accusé de réception de la requête; ensuite il reprend l'exécution sans attendre la fin de la fonction.

Il en résulte que les fonctions asynchrones ne devront avoir ni paramètres de sortie ni valeur de retour. Les routines sont toujours synchrones.

8.3.1 Routines

Les routines sont des fonctions internes aux scripts échangés.

Les routines seront déclarées dans le script échangé. Elles sont composées:

– d'une signature;

– de variables locales;

– d'un code programme.

Il ne sera pas possible de déclarer plus de 4096 routines dans un script échangé.

L'exécution d'une routine peut être déclenchée:

– par une instruction d'appel explicite **call (CALL)**, comprenant l'identificateur de fonction de la routine dans la partie opérande de l'instruction;

– lors de la réception d'une exception pendant une instruction **external call (XCALL)**, où la table de définition de filet mappe à l'identificateur de message de l'exception reçue, l'identificateur de fonction de la routine.

– lors de l'examen de la file d'attente des messages reçus, soit lorsque aucune routine ne s'exécute, soit lors de la rencontre d'une instruction **yield (YIELD)**,), où la table de définition de filet mappe à l'identificateur de message de l'exception reçue, l'identificateur de fonction de la routine ou bien se trouve être une opération **run** de l'API MHEG-3 en direction de la routine.

Les paramètres peuvent être passés aux routines selon les modes suivants:

– par **value (valeur)**: la valeur du type du paramètre est passée à la routine;

– par **reference (référence)**: un identificateur de donnée référençant une variable globale, une variable dynamique ou une constante dont le type est le même que celui du type du paramètre est passé à la routine.

Dans les deux cas, la valeur du paramètre transmis devient la valeur de la variable locale dont l'index correspond à l'index du paramètre. La variable locale correspondant à un paramètre passé par référence sera de type **data identifier**.

Les identificateurs de données des variables locales ne seront pas passés par référence.

8.3.2 Services

Les services sont des fonctions fournies par l'environnement d'exécution pouvant être invoquées par un script échangé.

Les services seront déclarés à l'intérieur du script échangé, comme une partie de déclaration de bloc, par:

– leur signature;

– leur nom global d'opération IDL.

On ne pourra pas déclarer plus de 256 services dans chaque bloc d'un script échangé.

On ne pourra pas déclarer plus de 192 blocs dans un script échangé.

Un service peut être appelé par une instruction **external call (XCALL)**.

Des paramètres peuvent être passés aux services selon l'un des modes suivants:

– **in (entrée)**: on passe au service un identificateur de donnée référençant une variable ou une constante dont le type est le même que celui du paramètre.

– **inout (entrée/sortie)**: on passe au service un identificateur de donnée référençant une variable dont le type est le même que celui du paramètre; lors du retour, la variable est mise à jour avec sa nouvelle valeur.

– **out (sortie)**: identique au précédent, cependant la valeur de la variable n'est pas utilisée par le service.

8.3.3 Fonctions prédéfinies

Les fonctions prédéfinies correspondent aux opérations fournies par l'interface du moteur MHEG-3.

Les fonctions prédéfinies ont des identificateurs de fonction prédéfinis et ainsi n'auront pas à être déclarées à l'intérieur d'un script échangé.

La présente Recommandation n'étant pas spécialement liée à l'une des Recommandations UIT-T de la série T.170 (et parties de l'ISO/CEI 13522), les opérations de l'API MHEG utilisées pour manipuler des objets MHEG ne sont donc pas clairement définies. Cependant, la présente Recommandation spécifie la procédure à utiliser pour définir une API MHEG et pour spécifier le mappage des opérations de cette API MHEG avec les identificateurs de fonctions prédéfinis. Ceci est décrit dans l'Annexe E.

De plus, la présente Recommandation définit l'API MHEG-3, à savoir l'interface fourni par les moteurs MHEG-3 pour la manipulation d'objets MHEG. Cette interface est décrit dans le paragraphe 15 et l'Annexe F. Cette interface peut être utilisé à partir des scripts auxquels on fera correspondre ensuite des identificateurs de fonctions prédéfinis. L'Annexe C contient la liste de ces fonctions prédéfinies avec leur identificateurs.

Les fonctions prédéfinies peuvent être appelées et recevoir des paramètres en utilisant les mêmes mécanismes que ceux utilisés pour les services.

8.4 Messages

La SIR MHEG définit deux types de messages:

– exceptions de bloc (voir 8.4.1);

– messages prédéfinis (voir 8.4.2).

Tous les messages auront une signature (ou prototype) formée d'une liste ordonnée de paramètres formels (membres) définis par leur type.

Tous les messages sont référencés par leur identificateur de message de manière unique et non ambiguë.

8.4.1 Exceptions de bloc

Les exceptions de bloc sont envoyées par l'environnement d'exécution à un *rt-script* à la suite de l'invocation d'un service par un *rt-script*.

Les exceptions de bloc seront déclarées dans les scripts échangés, comme une partie de la déclaration de bloc, par:

– leur signature;

– leur nom global d'exception IDL.

On ne pourra pas déclarer plus de 256 exceptions dans chaque bloc d'un script échangé.

8.4.2 Messages prédéfinis

Les messages prédéfinis envoyés à un *rt-script* peuvent être l'un des suivants:

– une exception de l'interface du moteur MHEG-3, à savoir l'API MHEG, soulevée par le moteur MHEG-3 suite à l'invocation par le *rt-script* d'une fonction prédéfinie;

– consécutif à l'invocation d'une opération de l'API MHEG-3 en direction du *rt-script;*

– l'exception **InstructionExecutionError**, soulevée consécutivement à l'apparition d'une erreur dans l'exécution d'une instruction du *rt*-script.

Les messages prédéfinis ont des identificateurs de message prédéfinis qui ne seront pas déclarés dans un script échangé.

La présente Recommandation n'étant pas spécialement liée à l'une des Recommandations UIT-T de la série T.170 (et parties de l'ISO/CEI 13522), les exceptions de l'API MHEG utilisées pour manipuler des objets MHEG ne sont donc pas clairement définies. Cependant, la présente Recommandation spécifie la procédure à utiliser pour définir une API MHEG et pour spécifier le mappage des exceptions de cette API MHEG avec les identificateurs de messages prédéfinis. Ceci est décrit dans l'Annexe E.

L'Annexe C spécifie la manière de faire mapper l'exception **InstructionExecutionError** et les messages résultant des opérations de l'API MHEG-3 avec les identificateurs de messages prédéfinis.

8.5 Instructions

La partie code programme des routines est composée d'une séquence d'instructions. A l'opposé du reste d'un script échangé qui est manipulé lors de la préparation du script, les instructions sont seulement traitées après la création d'un *rt-script*, lorsque leur routine d'appartenance est activée.

Une instruction sera formée d'un code opérateur (op-code) suivi de zéro ou plusieurs opérandes. Les nombres, type et codage des opérandes sont pleinement déterminés par le code opérateur.

Selon la règle, les opérandes complètent les instructions tandis que les valeurs des paramètres sont extraites de la pile de paramètres.

Le paragraphe 9 contient une description de la performance de l'unité d'exécution d'instructions tandis que le paragraphe 13 contient celle de la sémantique détaillée de chaque instruction.

8.6 Identificateurs

On utilise des identificateurs pour référencer les entités de la SIR MHEG, à savoir les types, données, fonctions et messages, de manière non ambiguë d'un bout à l'autre des scripts échangés.

8.6.1 Identificateurs de type

Les identificateurs de type (TID, *type identifiers*) seront codés sur deux octets de la manière suivante:

– les types primitifs et les types prédéfinis auront des TIDs prédéfinis comme ceux définis dans l'Annexe C;

– les types déclarés dont l'index (à partir de 0) dans la table de déclaration de type est X auront un TID de (X + 4000h).

Ainsi:

– les TIDs de 0 à 3FFFh référenceront des types prédéfinis;

– les TIDs de 4000h à 7FFFh référenceront des types déclarés.

8.6.2 Identificateurs de données

Les identificateurs de donnée (DID, *data identifiers*) seront codés sur deux octets de la manière suivante:

– les constantes dont l'index (à partir de 0) dans la table de déclaration de constantes est X auront un DID de X;

– les variables globales dont l'index (à partir de 0) dans la table de déclaration de variables globales est X auront un TID de (X + 1000h);

– les variables locales dont l'index (à partir de 0) dans la table de déclaration de variables locales est X auront un TID de (X + 8000h);

– les variables dynamiques auront un DID commençant à partir de 8100h. La procédure d'allocation des identificateurs de donnée aux variables n'étant pas définie; les moteurs MHEG-3 peuvent donc gérer divers schémas d'allocation.

Ainsi:

– les DIDs de 0 à 0FFFh référenceront des constantes;

– les DIDs de 1000h à 7FFFh référenceront des variables globales;

– les DIDs de 8000h à 80FFh référenceront des variables locales;

– les DIDs de 8100h à FFFFh référenceront des variables dynamiques.

8.6.3 Identificateurs de fonctions

Les identificateurs de fonction (FID, *function identifiers*) seront codés sur deux octets de la manière suivante:

– les routines dont l'index (à partir de 0) dans la table de déclaration de routines est X auront un FID de X;

– les fonctions prédéfinies dont l'index (à partir de 0) dans la table de déclaration de fonctions prédéfinies est X auront un FID de (X + 1000h);

– les services dont l'index (à partir de 0) dans une déclaration de bloc est X et dont l'index de bloc dans la table de déclaration de blocs est Y (à partir de 0) auront un FID de (((Y + 64) << 8) + X).

Ainsi:

– les FIDs de 0 à 0FFFh référenceront des routines;

– les FIDs de 1000h à 3FFFh référenceront des fonctions prédéfinies;

– les FIDs de 4000h à FFFFh référenceront des services.

8.6.4 Identificateurs de messages

Les identificateurs de message (MID, *message identifiers*) seront codés sur deux octets de la manière suivante:

– les messages prédéfinis dont l'index (à partir de 0) dans la table des messages prédéfinis est X auront un MID de X;

– les exceptions dont l'index (à partir de 0) dans une déclaration de bloc est X et dont l'index de bloc dans la table de déclaration de blocs est Y (à partir de 0) auront un MID de (((Y + 64) << 8) + X).

Ainsi:

– les MIDs de 0 à 3FFFh référenceront des messages prédéfinis;

– les MIDs de 4000h à FFFFh référenceront des exceptions de bloc.

9 La machine virtuelle SIR MHEG

Le présent paragraphe détaille la machine virtuelle SIR MHEG et plus précisément le modèle d'exécution du code SIR MHEG.

9.1 Structure de la machine virtuelle SIR MHEG

La machine virtuelle SIR MHEG est formée d'un ensemble de composants logiques abstraits. Sa description a pour objectif de clarifier la sémantique opérationnelle du code SIR MHEG.

Un moteur MHEG-3 aura le même comportement dans l'interprétation du code SIR MHEG que celui de la machine virtuelle décrite. Il interprétera des déclarations et des instructions de SIR MHEG afin de produire des effets externes identiques à tous les égards.

Cependant, cela n'implique pas de contraintes sur la technologie ou l'organisation pouvant être effectivement utilisée pour implémenter un moteur MHEG-3. Il n'est pas nécessaire d'implémenter un interpréteur réel de script selon la description de la machine virtuelle à partir du moment où celui-ci fournit des fonctionnalités équivalentes.

La machine virtuelle SIR MHEG est composée de:

– zones de mémoire (voir 9.3);

– unités de traitement (voir 9.5).

On associe certaines zones de mémoire à un *mh-script* et ainsi tous les *rt-scripts* qui sont créés à partir de ce dernier les partagent. On associe d'autres zones de mémoire à chaque *rt-script*.

Les unités de traitement sont affectées à un *rt-script*. Cependant, un moteur MHEG-3 peut exécuter plusieurs *rt-scripts* en parallèle. Dans ce cas, il maintiendra un contexte d'exécution séparé pour chaque *rt-script*.

NOTE – En d'autres termes, la machine virtuelle SIR MHEG est monotâche. On peut dérouler des applications multitâches en associant un *rt-script* distinct à chaque tâche.

9.2 Structures et notations

9.2.1 Table

Une table **T** est composée d'un tableau d'entrées homogènes **T[i]** accessibles par leur indice **i**. Ces entrées ont la même structure mais pas obligatoirement la même taille. Les entrées sont constituées d'un ou plusieurs champs **fld**. Certaines entrées peuvent être void. Les indices sont des identificateurs SIR MHEG, à savoir des valeurs consécutives prises dans un rang donné et ne commençant pas obligatoirement par 0 pour une table donnée. Le mécanisme d'accès sous-jacent (indexage séquentiel, accès direct, hachage, etc.) n'est pas spécifié. La notation utilise les primitives suivantes pour exprimer une manipulation dans une table **T**:

– **T[i]**pour accéder à l'entrée **i**;

– **T[i]** = **VAL** pour affecter **VAL** à l'entrée **i**;

– **T[i].fld** pour accéder au champ **fld** de l'entrée **i**;

– **T[i].fld** = **VAL** pour affecter la valeur **VAL** au champ **fld** de l'entrée **i**.

9.2.2 Pile

Une pile est composée d'un tableau d'éléments homogènes. Les éléments sont insérés en sommet de pile. Seul l'élément du haut (dernier entré) peut être accédé à tout moment. Lorsqu'on l'ôte de la pile, on le perd et l'élément suivant devient le sommet de pile. La notation utilise les primitives suivantes pour exprimer la manipulation de la pile d'appel **CS**:

– **CS.push(F)**: insère la trame **F** en sommet de pile, incrémente le registre de pointeur de pile (FP, *frame pointer*);

– **CS.pop()**: décrémente FP, ôte la trame de sommet de pile et ensuite la retourne;

– **CS[FP]**: retourne la valeur de la trame de sommet de pile.

Comme pour les tables, la notation "." est utilisée pour accéder aux champs d'éléments de pile.

9.2.3 Pile de paramètres

La pile de paramètres est un cas particulier car c'est une pile de caractères (non typée) utilisée pour stocker les valeurs typées. La notation utilise les primitives suivantes pour exprimer la manipulation de la pile de paramètres **PS**, dans laquelle **tid** est l'identificateur de type d'un type primitif comme indiqué dans la Table 1.

– **PS.push(VAL)**: insère la valeur **VAL** en sommet de pile;

– **PS.pop(tid)**: ôte la valeur en sommet de pile dont l'identificateur est **tid**, ensuite la retourne;

– **PS[SP](tid)**: retourne la valeur de la valeur du sommet de pile dont l'identificateur de type est **tid**.

9.2.4 File d'attente

Une file d'attente est composée d'un tableau d'éléments homogènes. Les éléments sont insérés en fin de file d'attente. Seul l'élément du début (premier entré) peut être accédé à tout moment. Lorsqu'il est ôté de la file d'attente, il est perdu et l'élément suivant est placé en début de file d'attente. La notation utilise les primitives suivantes pour exprimer la manipulation de la file d'attente de message **MQ**:

– **MQ.insert(M)**: insère la valeur du message **M** en fin de file d'attente;

– **MQ.remove()**:incrémente le registre de pointeur de file d'attente (QP, *queue pointer*), ôte l'élément en début de file d'attente et le retourne;

– **MQ[QP]**: retourne la valeur de l'élément en début de file d'attente.

Comme pour les tables, la notation "." est utilisée pour accéder aux champs d'éléments de file d'attente.

9.2.5 Représentation des données

La représentation des données et des structures dépend de l'implémentation. Bien qu'ils puissent représenter chaque valeur de type de donnée sur un nombre minimal de caractères, les interpréteurs de scripts ne sont pas obligés de le faire. Le Tableau 2 fait état de ce nombre minimal.

Tableau 2/T.173 – Nombre minimal de caractères nécessaires à la représentation des valeurs de types

Type	Nombre minimal de caractères nécessaires à la représentation des valeurs de types
octet	1
short	2
long	4
unsigned short	2
unsigned long	4
float	4
double	8
boolean	1
character	2 (1 pour les jeux de caractères restreints)
data identifier	2
object reference	dépend de l'implémentation
string	taille de caractère x (longueur de la chaîne + 1)
sequence	taille du type d'élément x longueur de la séquence (nombre réel d'éléments) + 2
array	taille du type d'élément x taille du tableau
structure	somme des tailles des types d'éléments
union	taille du type d'élément " le plus grand " + 1
type identifier	2
function identifier	2
message identifier	2
identificateur de bloc	1

NOTE – La notation ne distingue pas les valeurs des longueurs fixes de celles des longueurs variables. Les interpréteurs de script peuvent stocker des valeurs de longueur variable sur le tas (*heap*). On utilise **VT[i].val** pour accéder à la valeur d'une variable même si en fait on pourrait la stocker dans la table de variables comme un pointeur sur le tas.

Lorsque **VAL** est une valeur de type construit, la notation de l'accès à ses éléments se fait comme suit:

– **VAL.tag**: étiquette d'union;

– **VAL.val**: valeur d'union;

– **VAL[n]**: valeur du énième élément d'une séquence, d'une chaîne, d'une structure ou d'un tableau;

– **VAL.lg**: longueur effective d'une séquence ou d'une chaîne.

Les sémantiques d'exécution sont exprimées en pseudo-C. Les expressions entre des quotes simples indiquent la valeur allouée, par exemple "void" indique que TID a pour valeur 0.

9.3 Zones de mémoire

Dans la machine virtuelle SIR MHEG, les zones de mémoire sont utilisées pour stocker toutes les informations nécessaires à l'interprétation d'un script échangé particulier.

Les zones de mémoire peuvent être associées soit à un *mh-script* (voir 9.3.1) ou à un *rt-script* (voir 9.3.2).

9.3.1 Zone de mémoire de *mh-script*

Les zones de mémoire de *mh-script* devraient être totalement chargées lors de la phase de chargement, à savoir lors de l'initialisation d'un *mh-script*. Tous les *rt-scripts* dépendants de ce *mh-script* devraient pouvoir y accéder pour en faire usage. Les zones de mémoire de *mh-script* ne seront pas modifiées pendant la phase d'exécution jusqu'à la destruction du *mh-script*, sauf en cas de spécification contraire (tel est le cas de la table de définition de bloc). Les zones de mémoire des *mh-scripts* incluent:

– des zones de données (voir 9.3.1.1);

– des zones de code (voir 9.3.1.2).

9.3.1.1 Zones de données

Les zones de données servent à stocker les définitions et les valeurs des données globales de script. Les zones de données comprennent:

– la table des définitions de types (TT) (voir 9.3.1.1.1);

– la table des constantes (CT, *constant table*) (voir sous-clause 9.3.1.1.2);

– la tables des définitions de variables globales (GT, *global table*) (voir 9.3.1.1.3).

9.3.1.1.1 Table des définitions de types

La table des définitions de types mappe leur description aux types définis de script, représentés par des identificateurs de types:

– **TT[TID].val**: description du type.

NOTE – La représentation utilisée pour la description de type n'est pas spécifiée; cependant, celle-ci devrait permettre de vérifier facilement l'appartenance d'une valeur à tel ou tel type.

9.3.1.1.2 Table des constantes

La table des constantes mappe leur type et leur valeur aux constantes de script, représentées par des identificateurs de types:

– **CT[DID].TID**: type de la constante (exprimé comme un identificateur de type);

– **CT[DID].val**: valeur de la constante (dépend de son type).

9.3.1.1.3 Table des définitions de variables globales

La table des définitions de variables globales mappe leur type et leur valeur initiale aux variables globales de script, représentées par des identificateurs de types:

– **GT[DID].TID**: type de la variable globale (exprimé comme un identificateur de type);

– **GT[DID].val**: valeur initiale de la variable globale (dépend de son type).

9.3.1.2 Zones de code

Les zones de code s'utilisent pour stocker les adresses et le code programme des fonctions de scripts. Les zones de code comprennent:

– la table des définitions de routines (RT, *routine table*) (voir 9.3.1.2.1);

– la table des définitions de blocs (PT, *package table*) (voir 9.3.1.2.2);
– la table des définitions de services (ST, *service table*) (voir 9.3.1.2.3);
– la table des définitions d'exceptions (XT, *exception table*) (voir 9.3.1.2.4);
– la table des définitions de filets (HT, *handler table*) (voir 9.3.1.2.5);
– La zone de programme code, formée de la séquence d'instructions de chaque routine (voir 9.3.1.2.6).

9.3.1.2.1 Table des définitions de routines

La table des définitions de routines mappe leur description de signature, leurs déclarations de variables locales et leur code programme aux routines de script, représentées par des identificateurs de fonction:

a) **RT[FID].TID**: type de la valeur de retour (exprimé comme un identificateur de type);
b) **RT[FID].nbp**: nombre de paramètres;
c) **RT[FID].sig**: description de signature où:
   1) **RT[FID].sig[i].TID** est son type (exprimé comme un identificateur de type) du énième paramètre
   2) **RT[FID].sig[i]**.mod est son mode de passage (**value** ou **reference**) du énième paramètre;
d) **RT[FID].LT**: déclaration des variables locales à la routine (dont les **nbp** éléments sont les paramètres effectifs de la routine);
e) **RT[FID].IP**: pointeur sur la première instruction du code de la routine.

Les variables locales détenant des paramètres passés par **reference** auront un **data identifier** identique à leur type, tandis que les variables locales détenant des paramètres passés par **value** auront le même type que celui de la description de signature du paramètre correspondant.

9.3.1.2.2 Table des définitions de blocs

La table des définitions de blocs mappe leur nom de bloc et de l'information complémentaire aux blocs définis dans le script, représentés par des identificateurs de bloc (PID, *package identifer*) tels que déclarés dans la table de déclaration de blocs de la SIR MHEG:

– **PT[PID].name**: nom du bloc;
– **PT[PID].nbf**: nom des services dans le bloc;
– **PT[PID].nbm**: nombre d'exceptions définies dans le bloc;
– **PT[PID].sts**: état en cours du bloc (**unchecked**, **available**, **ready**, **opened**);
– **PT[PID].or**: référence à objet initiale du bloc.

Le statut initial d'un bloc est **unchecked**. Il devient **available** une fois que la procédure **package availability** a été effectuée avec succès. Il devient ensuite **ready** une fois que la procédure **package load** a été effectuée avec succès. Finalement, il est **opened** lorsqu'une référence à objet initiale et valide au bloc a été stockée dans le champ **PT[PID].or**, en vue d'une utilisation dans les invocations ultérieures des services.

Une exception à cette règle établit au 9.3.1 que,

– Les champs **PT[PID].sts** peuvent être modifiés pendant l'exécution à chaque changement d'un status de bloc;
– Les champs **PT[PID].or** peuvent être modifiés pendant l'exécution lors d'un chargement de bloc.

9.3.1.2.3 Table des définitions de services

La table des définitions de services mappe leur description de signature et leur nom global d'opération IDL aux services externes définis dans le script, représentés par des identificateurs de fonction de la SIR MHEG:

a) **ST[FID].TID**: type de valeur de retour (exprimé par un identificateur de type);

b) **ST[FID].syn**: mode d'appel (**synchronous**, **asynchronous**);

c) **ST[FID].nbp**: nombre de paramètres;

d) **ST[FID].sig**: description de signature, où:

   1) **ST[FID].sig[i].TID** est le type (exprimé par un identificateur de type) du énième paramètre;

   2) **ST[FID].sig[i].mod** est le mode de passage (**in**, **inout** or **out**) du énième paramètre;

e) **ST[FID].name**: nom global IDL de l'opération invoquée par le service.

La spécification de mappage spécifique de la plate-forme sera utilisée pour mapper le **ST[MID].name** à un nom spécifique de la plate-forme.

9.3.1.2.4 Table des définitions d'exceptions

La table des définitions d'exceptions mappe leur description de signature et leur nom global d'opération IDL aux messages définis dans le script échangé, représentés par des identificateurs de message:

– **XT[MID].name**: nom global IDL de l'exception provoquant le message;

– **XT[MID].nbm**: nombre de membres;

– **XT[MID].sig**: description de signature où **XT[MID].sig[i].TID** est le type (exprimé comme un identificateur de type) du énième membre.

La spécification de mappage spécifique à la plate-forme sera utilisée pour mapper le **XT[MID].name** à un nom spécifique à la plate-forme.

9.3.1.2.5 Table des définitions de filets

La table des définitions de filets mappe leurs routines aux messages représentés par des identificateurs de fonctions:

– **HT[MID].FID**: identificateur de la routine à invoquer pour manipuler le message.

Si on mappe une routine à un message dans la table du filet, la signature de cette routine doit correspondre à la signature de ce message. La correspondance entre les signatures sera vérifiée lors du chargement et les entrées ne correspondant pas seront rejetées.

La table des définitions de filet est utilisée par l'unité d'exécution du *rt-script*. Lorsque l'unité d'exécution du *rt-script* ôte un message de la file d'attente de messages, celui-ci invoque la routine correspondant au message en y mettant les paramètres du message comme paramètres de la routine.

9.3.1.2.6 Zone de code programme

Une instruction est composée d'un code opérateur d'une longueur de 1 octet suivi de zéro à trois opérandes. La nature du code opérateur permet de déterminer le nombre et la longueur de ses opérandes (voir la table d'instructions). Les opérateurs et les opérandes sont tous les deux codés de manière optimisée pour en faciliter la permutation.

NOTE – Un interpréteur de script (particulièrement sur les machines 32 bits) peut aligner les instructions lors du chargement, à savoir insérer des octets de bourrage pour représenter chaque instruction sur quatre octets; cela rend plus facile l'incrémentation du pointeur d'instruction. L'interpréteur de script a aussi la possibilité de laisser les instructions en bloc et de déterminer le nombre d'octets à ajouter lors de l'exécution.

9.3.2 Zones de mémoire du *rt-script*

Les zones de mémoire du *Rt-script* sont initialisées lors de la création de ce dernier et peuvent être modifiées pendant son exécution. Les zones de mémoire du *rt-script* comprennent:

– des zones de mémoire dynamique (voir 9.3.2.1);

– des registres (voir 9.3.2.2).

9.3.2.1 Zones de mémoire dynamique

Les zones de mémoire dynamique s'utilisent pour représenter les données et le contexte courant d'exécution du *rt-script*.

Elles comprennent:

– la table de variables (VT, *variable table*) (voir 9.3.2.1.1);

– la pile d'appel (CS, *call stack*) (voir 9.3.2.1.2);

– la pile de paramètre (PS, *parameter stack*) (voir 9.3.2.1.3);

– la file d'attente de messages (MQ, *message queue*) (voir 9.3.2.1.4);

– le tas (voir 9.3.2.1.5).

9.3.2.1.1 Table des variables

La table des variables mappe leurs types et leurs valeurs courantes aux variables des *rt-scripts*, représentées par des identificateurs.

– **VT[DID].TID**: type de la variable (exprimé sous la forme d'un identificateur de type);

– **VT[DID].val**: valeur courante de la variable (dépendante du type).

La table des variables est initialisée lors de la création du *rt-script*. Elle comprend deux sous-tables:

– une copie de la table des variables globales associées au *mh-script;*

– la table des variables locales de la routine en cours d'exécution.

Les champs **VT[DID].val** sont modifiés à chaque exécution d'instruction d'affectation de variable.

Lorsqu'on change de routine (à la suite des instructions **CALL**, **RET** ou **YIELD**), la table des variables locales de la routine en cours est stockée et remplacée dans le VT par la table des variables locales de la nouvelle routine. Les premières entrées d'une table de variables locales constituent les paramètres passés à la fonction.

9.3.2.1.2 Pile d'appel

La pile d'appel sert à stocker le contexte courant d'invocation.

La pile d'appel est un tableau de trames d'appel. Chaque trame correspond au contexte du temps d'invocation d'une fonction active (routine, fonction externe ou action MHEG). Les trames sont stockées dans la pile d'appel par ordre d'invocation. La trame du haut de la pile d'appel, si elle existe, décrit le contexte d'exécution de la routine ayant appelé la fonction en cours d'exécution.

Chaque trame comprend les éléments suivants:

– **CS[i].FID:** identificateur de fonction de l'appelant;

– **CS[i].IP:** pointeur vers l'instruction à exécuter lors du retour d'exécution de la fonction en cours;

– **CS[i].LT:** table des variables locales de l'appelant (au moment de l'invocation);

– **CS[i].SP:** pointeur vers le sommet de la pile des paramètres (au moment de l'invocation).

Le champ LT d'une trame d'appel possède la structure d'une table de variables.

– **CS[i].LT[DID].TID:** identificateur de type de la variable dont l'identificateur est **DID**;

– **CS[i].LT[DID].val:** valeur de la variable.

La pile d'appel est modifiée par certaines instructions de contrôle de flux. A l'origine, la pile d'appel est vide. Lors de l'invocation d'une fonction, une trame décrivant cet appel sera empilée sur la pile d'appel. Lors d'un retour de fonction, cette trame sera dépilée de la pile d'appel. L'adresse de la trame du sommet de la pile d'appel sera stockée de manière permanente dans le registre FP.

9.3.2.1.3 Pile des paramètres

La pile des paramètres est utilisée pour stocker et retourner des valeurs d'instructions. La pile des paramètres est un tableau de valeurs de données. Le type de la valeur de donnée est déterminé par la séquence d'opérations ayant pour effet d'empiler la valeur sur la pile.

La pile des paramètres est utilisée par l'unité d'exécution d'instructions de la SIR MHEG. A l'origine, la pile d'appel est vide. Elle est modifiée par la plupart des instructions (opérateurs arithmétiques, opérateurs logiques, opérations de comparaison, manipulations de pile, affectations de variable, sauts conditionnels, appels). Lors de l'exécution d'une instruction, celle-ci dépile ses paramètres de la pile d'appel et empile sa valeur de retour dans la pile des paramètres. L'adresse de la trame du sommet de la pile d'appel est stockée de manière permanente dans le registre SP (SP, *stack pointer*).

9.3.2.1.4 File d'attente des messages

La file d'attente des messages est utilisée pour bufferiser les messages reçus par l'interpréteur de script. Chaque élément de la file d'attente comprend les éléments suivants:

– **MQ[i].MID:** identificateur de message;

– **MQ[i].LT:** liste des paramètres du message.

Le champ LT d'une file d'attente de messages aura la structure d'une table de variables:

– **MQ[i].LT[j].TID:** identificateur de type du jème paramètre;

– **MQ[i].LT[j].val:** valeur du jème paramètre.

Les messages seront insérés dans la file d'attente des messages de manière asynchrone par l'interpréteur de script au fur et à mesure de leur génération par l'environnement externe. La file d'attente des messages sera traitée par l'unité d'exécution de *rt-script* lorsque:

– le *rt-script* n'est pas en cours d'exécution, à savoir il n'y a pas de routine en cours d'exécution;

– ou bien sur rencontre d'une instruction **YIELD**.

Le début de la file d'attente (message suivant à empiler) sera stocké de manière permanente dans le registre QP. A l'origine, la file d'attente des messages est vide.

9.3.2.1.5 Tas

Le tas sert à stocker des variables dynamiques, représentées par des identificateurs de données:

– **VT[DID].TID:** type de la variable (exprimé sous la forme d'un identificateur de type);

– **VT[DID].val:** valeur courante de la variable (dépendante du type).

Les variables dynamiques sont référencées par des pointeurs d'un type opaque dont on ne connaît pas la représentation. On mappe en interne les identificateurs de données à ces pointeurs pour permettre l'accès à ces variables dynamiques d'une manière identique aux autres variables.

Les champs **VT[DID].val** sont modifiés lors de chaque affectation de variable par l'exécution d'une instruction d'affectation de variable.

L'application est responsable de l'allocation explicite et de la désallocation des variables dynamiques en utilisant les instructions **ALLOC** et **FREE**.

NOTE – Les interpréteurs de script peuvent aussi utiliser le tas pour stocker les valeurs des variables locales ou globales dont la longueur est de type variable. Dans ce cas, la table contiendra un pointeur sur le tas au lieu de la donnée elle-même.

9.3.2.2 Registres

Dans les registres sont consignés les états spécifiques de la machine virtuelle. Ceux-ci ont besoin d'être fréquemment modifiés pendant l'exécution d'un *rt-script*.

Les registres mis à jour par la machine virtuelle SIR MHEG sont:

– le pointeur d'instruction (IP, *instruction pointer*) ou pointeur de programme (voir 9.3.2.2.1);

– le pointeur de trame (FP, *frame pointer*) (voir 9.3.2.2.5);

– le pointeur de pile (SP) (voir 9.3.2.2.4);

– le pointeur de file d'attente (QP) (voir 9.3.2.2.6);

– le registre d'instruction (IR, *instruction register*) (voir 9.3.2.2.2);

– le registre d'erreur (ER, *error register*) (voir 9.3.2.2.3);

– le registre de fonction (FR, *function register*) (voir 9.3.2.2.7).

On ne spécifie pas la représentation des données détenues par les registres pointeurs. Tous les registres seront initialisés à la valeur null. Cette valeur n'est pas spécifiée.

9.3.2.2.1 Registre de pointeur d'instruction

Le registre IP pointe sur l'instruction suivante à exécuter dans un code programme de routine. Ce registre sera modifié par l'unité d'exécution de *rt-script* et par l'unité d'exécution d'instructions SIR MHEG comme faisant partie de l'exécution des instructions

9.3.2.2.2 Registre d'instruction

Le registre IR contient le code de l'instruction en cours d'exécution. Ce registre sera mis à jour par l'unité d'exécution de *rt-script* lors de chaque chargement de nouvelle instruction et sera accessible par l'unité d'exécution d'instructions de la SIR MHEG.

NOTE – Le registre IR n'aura pas besoin d'une longueur supérieure à 4 octets, mais on n'en spécifie pas la taille réelle.

9.3.2.2.3 Registre d'erreur

Le registre d'erreur (ER) contient le code de la dernière erreur rencontrée pendant l'exécution d'une instruction. Ce registre sera mis à jour par l'unité d'exécution d'instructions de la SIR MHEG à chaque rencontre d'erreur. La valeur null indique qu'aucune erreur n'a été rencontrée jusqu'à présent pendant l'exécution du *rt-script*.

Les codes d'erreur sont prédéfinis. Les codes d'erreur soulevés par chaque instruction sont définis dans le paragraphe 13.

Lorsqu'une erreur est rencontrée pendant l'exécution d'une instruction, le registre ER est mis à une valeur non nulle et une exception **InstructionExecutionError** apparaît. Ceci a pour effet d'insérer le message correspondant dans la file d'attente des messages.

9.3.2.2.4 Registre de pointeur de pile

Le registre SP pointe sur le sommet de la pile des paramètres. La valeur de ce registre sera mise à jour par l'unité d'exécution d'instructions de la SIR MHEG de la manière suivante:

– il sera incrémenté lors de chaque empilement de donnée sur la pile de paramètres;
– il sera décrémenté lors de chaque dépilement de donnée de la pile des paramètres.

9.3.2.2.5 Registre de pointeur de trame

Le registre FP pointe sur la trame du sommet de la pile d'appel. La valeur de ce registre sera mise à jour par l'unité d'exécution d'instructions de la SIR MHEG de la manière suivante:

– il sera incrémenté chaque fois qu'une fonction est appelée;
– il sera décrémenté chaque fois qu'une fonction est retournée.

9.3.2.2.6 Registre de pointeur de file d'attente

Le registre QP pointe sur le message suivant à ôter de la file d'attente. La valeur de ce registre sera décrémentée par l'interpréteur de script chaque fois qu'un message sera ôté.

9.3.2.2.7 Registre de fonction

Le registre FR contient le FID de la fonction en cours d'exécution. La valeur de ce registre sera mise à jour par l'interpréteur de script lors de chaque appel ou retour de fonction.

9.4 Etats des scripts

9.4.1 Etat d'un mh-script

L'état d'un *mh-script* sera soit **available** ou **not available**.

9.4.1.1 Non disponible (Not available)

L'état d'un *mh-script* sera dit **not available** dans l'un des cas suivants:

– l'initialisation du *mh-script* (à savoir, l'effet de l'opération **prepare** de l'API MHEG-3) n'est pas terminée;
– la destruction du *mh-script* (à savoir, l'effet de l'opération **destroy** de l'API MHEG-3) a été demandée

9.4.1.2 Disponible (Available)

L'état d'un *mh-script* est dit **available** si son initialisation s'est terminée avec succès et si sa destruction n'a pas été demandée.

Ceci implique que:

– l'analyse syntaxique du script échangé a été effectuée et les zones de mémoire correspondantes du *rt-script* ont été remplies;
– les blocs référencés dans le *mh-script* sont disponibles et ont été chargés avec la procédure **package load**.

9.4.2 Etats d'un rt-script

L'état d'un *rt-script* pourra être l'un des suivants: **not ready**, **ready**, **running, erroneous**.

9.4.2.1 Non prêt (Not ready)

L'état d'un *rt-script* sera dit **not ready** dans l'un des cas suivants:

– l'initialisation du *rt-script* (à savoir, l'effet de l'opération **new** de l'API MHEG-3) n'est pas terminée;
– la destruction du *rt-script* (à savoir, l'effet de l'opération **delete** de l'API MHEG-3) a été demandée.

A l'origine, l'état d'un *rt-script* est à **not ready.** Autrement, il est mis à **not ready** lorsqu'une opération **delete** est invoquée sur ce *rt-script*.

9.4.2.2 Prêt (Ready)

L'état d'un *rt-script* sera dit **ready** si toutes les conditions suivantes sont réunies:

– l'initialisation du *rt-script* s'est terminée avec succès sur ce *mh-script*;

– aucune requête de destruction n'a été demandée sur ce *rt-script*;

– le registre IP contient la valeur "null", il n'y a donc pas de routine en cours d'exécution**;**

– le registre ER contient la valeur "null".

Ceci implique que la pile d'appel, la file d'attente des messages et la pile des paramètres sont vides.

Cependant, il n'est pas nécessaire que les valeurs de variables globales soient à la même valeur initiale; une fois qu'il ne reste plus d'instructions à exécuter et plus de message dans la file d'attente, un *rt-script* revient à l'état **ready**.

L'état d'un *rt-script* change

– de **not ready** à **ready** lorsqu'une opération **new** est invoquée sur le *rt-script*;

– de **running** à **ready** lorsque l'unité d'exécution du *rt-script* n'a plus d'instruction à exécuter ou bien lorsqu'on a invoqué une opération **stop** ou **reinit**;

– de **erroneous** à **ready** lorsqu'on a invoqué une opération **stop** ou **reinit.**

9.4.2.3 En cours d'exécution (Running)

L'état d'un *rt-script* sera dit **running** si toutes les conditions suivantes sont réunies:

– l'initialisation du *rt-script* s'est terminée avec succès sur ce *mh-script*;

– aucune requête de destruction n'a été demandée sur ce *rt-script*;

– le registre IP contient la valeur "null", il n'y a donc pas de routine en cours d'exécution;

– le registre ER contient la valeur "null".

L'état d'un *rt-script* change de **ready** à **running** lorsqu'un message est présent dans la file d'attente des messages et que l'unité d'exécution de *rt-script* est activée. Cela peut être le résultat de l'invocation d'une opération **run**.

9.4.2.4 Erroné (Erroneous)

L'état d'un *rt-script* sera dit **erroneous** si la valeur du registre ER est différente de null, c'est-à-dire qu'une erreur est apparue pendant l'exécution du *rt-script.*

L'état d'un *rt-script* passe de **running** à **erroneous** lorsqu'une erreur d'exécution d'instruction est détectée par l'unité d'exécution des instructions *rt-script*.

9.5 Unités de traitement

Le présent sous-paragraphe décrit le flux des commandes de la machine virtuelle SIR MHEG et la sémantique de ces instructions.

Pour les besoins de description de la machine virtuelle, le processus principal de la machine virtuelle et les unités actives d'exécution de *rt-script* sont supposés fonctionner en parallèle. On ne spécifie pas l'ordonnancement des diverses tâches.

9.5.1 Réception de message

Le processus principal de l'interpréteur de script reçoit et manipule des événements. En leur absence, il est dans l'état **idle.** Les événements reçus par l'interpréteur de script peuvent être:

– des invocations d'opérations de l'API MHEG-3;

– des messages correspondant aux occurrences d'exceptions soulevées comme résultat de l'invocation d'un service ou d'une fonction prédéfinie.

9.5.1.1 Opérations de l'API MHEG-3

Les opérations de l'API MHEG-3 peuvent être invoquées par une unité d'exécution de *rt-script*, par un autre composant du moteur MHEG-3 ou par des processus externes au moteur MHEG-3.

Lorsqu'on invoquera une opération de l'API MHEG-3, le processus principal se déroulera selon la sémantique de l'API MHEG-3 décrite dans le paragraphe 15.

9.5.1.2 Exception externe

Dans le cas de l'émission d'un message vers un *rt-script*, provenant soit de l'interpréteur d'actions (exception de l'API MHEG) ou de l'environnement d'exécution, si ce message correspond en fait à une exception soulevée par l'API MHEG ou par l'environnement d'exécution comme étant la conséquence de l'invocation d'une opération résultant d'une instruction **XCALL** effectuée par ce script, le processus principal effectuera l'analyse syntaxique des paramètres d'exception et construira une structure de message. Celle-ci sera composée de l'identificateur de message de l'exception suivi de ses membres effectifs (le premier étant la référence à objet de l'objet d'origine). Ensuite,

– si l'exception est le résultat de l'invocation d'une opération synchrone en cours d'exécution, le processus principal demandera à l'unité d'exécution de *rt-script* de terminer l'instruction **XCALL** (ensuite de dépiler sa trame de la pile d'appel) sans rechercher les paramètres de sortie ou la valeur de retour, puis de déclencher immédiatement après, la routine correspondant à l'identificateur du message d'exception, en y incluant ses membres (paramètres formels).

  L'effet pour cette routine sera identique à celui qu'aurait produit son invocation par une instruction **CALL**;

– si l'exception est le résultat de l'invocation d'une opération synchrone précédemment terminée (que ce soit avec succès ou non), la processus principal ignorera l'exception;

– si l'exception est le résultat de l'invocation d'une opération asynchrone, le processus principal insérera le message construit dans la file d'attente des messages du *rt-script* destinataire.

9.5.1.3 Erreur d'exécution d'instruction (Exception **InstructionExecutionError**)

Lorsque l'exception interne **InstructionExecutionError** apparaîtra, le processus principal construira un message composé de l'identificateur du message correspondant à l'exception, suivi d'un membre mis à la valeur du registre d'erreur, puis insérera dans la file d'attente des messages du *rt-script* dont l'exécution a produit l'exception.

9.5.1.4 Exception de l'API MHEG-3

Lorsqu'une exception résultant de l'invocation d'une opération de l'API MHEG-3 sera renvoyée à un *rt-script*, le processus principal construira un message composé de l'identificateur du message correspondant à l'exception, suivi de ses membres, puis insérera dans la file d'attente des messages du *rt-script* dont l'exécution a produit l'exception.

9.5.2 Initialisation de mh-script

Sur invocation de l'opération **prepare** de l'API MHEG-3, l'interpréteur de script accédera au flux ou au fichier en utilisateur l'identificateur du système fourni et effectuera l'analyse syntaxique du script. Ensuite, l'interpréteur

– analysera la syntaxe de la partie déclarations et initialisera CT, GT, TT, RT, ST, PT, XT, HT et **RT[i].LT** pour chaque routine **i**; ceci incluant les vérifications appropriées (vérification de filet, procédure **package availability**);

– analysera la syntaxe de la partie instructions pour remplir la zone de code programme de chaque routine;

– effectuera la procédure **package load**, pour établir les liens statiques avec les blocs selon les spécifications de mappage de la plate-forme;

– mettra le *mh-script* à l'état **available.**

NOTE – La sémantique de chargement de bloc doit être définie par la spécification de mappage de la plate-forme. Il se peut que le moteur MHEG-3 prenne la responsabilité d'optimiser sa stratégie de gestion de ressources, à savoir en déchargeant temporairement des blocs afin de libérer de la mémoire, ou en chargeant les blocs seulement lorsque les *rt-scripts* sont créés et même seulement lorsque les services sont invoqués.

9.5.3 Initialisation de rt-script

Sur invocation de l'opération **new** de l'API MHEG-3, l'interpréteur de script créera un contexte pour le *rt-script* destinataire. Ainsi l'interpréteur de script:

– initialisera les zones de mémoire dynamique;

– initialisera tous les registres à la valeur null;

– créera une unité d'exécution de *rt*-script pour le *rt-script* considéré;

– mettra le *rt-script* à l'état **ready**.

9.5.4 Unité d'exécution de rt-script

Lorsqu'elle sera activée et tant qu'on ne lui aura pas dit d'arrêter le *rt-script* en cours, l'unité d'exécution du *rt-script* se déroulera comme suit:

```
rt-script-execution-unit ()
{
FID fid = 'null';
if (IP == 'null')                              // no next instruction
{
        while (fid == 'null')
        {
                if (QP == 'null') then exit;                    // return
                fid= HT[MQ[QP].MID].FID;            // find handler for message
                if (fid != 'null') then                         // handler found
                {
                        CS.push({IP, FR, SP, 'null'});      // stack routine call
                        FR = fid;
                        IP = RT[FR].IP;     // branch to start of routine
                }
                MQ.remove();                            // remove message
        }
}
// endif

while (IP != 'null'):
{
```

```
        IR = *IP++;                 // load next instruction and increment program counter
        instruction-execution-unit();       // call the MHEG-SIR instruction execution unit
}
// endwhile
return;                                     // return to script interpreter
}
```

9.5.5 Unité d'exécution d'instructions SIR MHEG

Lorsqu'elle sera appelée par l'unité d'exécution de *rt-script,* l'unité d'exécution d'instructions SIR MHEG d'un *rt-script* décodera le code opérateur contenu dans le premier octet de IR, ensuite interprétera l'instruction correspondant à ce code opérateur comme il est spécifié au paragraphe 13 et enfin retournera à l'unité d'exécution de *rt-script.*

L'unité d'exécution d'instructions dépile en cas de besoin de la pile des paramètres ceux qui sont utilisés pour effectuer l'instruction. Elle empile en cas de besoin sur la pile des paramètres ceux qui sont produits par le résultat de l'instruction.

Le Tableau 3 résume les effets des instructions sur les divers composants de la machine virtuelle SIR MHEG définie dans le présent paragraphe.

10 Dispositions pour l'accès à un environnement d'exécution

Le présent paragraphe décrit les mécanismes définis dans la présente Recommandation permettant aux *rt-scripts* d'accéder à, et d'échanger des données avec les fonctions fournies sur la plate-forme par l'environnement d'exécution.

10.1 Modèle général

L'interface, fourni par les logiciels externes disponibles dans l'environnement d'exécution, doit être déclaré dans le script échangé en tant que partie de la déclaration de bloc, de sorte que l'interpréteur de script sache comment accéder à cette interface lorsqu'il est invoqué par le script.

Une déclaration de bloc décrit un ensemble de services (fonctions externes) sous la forme de signatures comprenant le type et le mode de passage de chaque paramètre.

La SIR MHEG spécifie dans les scripts échangés le mode d'expression des fonctions externes appelantes, des paramètres à passer, des valeurs de retour à extraire et des exceptions externes.

La présente Recommandation spécifie aussi comment ces expressions seront interprétées par les moteurs MHEG-3.

La présente Recommandation traite aussi l'échange (à savoir l'appel de fonctions, le passage de paramètres, la récupération de valeurs de retour et la manipulation d'expressions) entre un moteur MHEG-3 et l'environnement d'exécution. Dans cette optique, la présente Recommandation contient des dispositions spécifiant comment les logiciels externes devraient fournir aux moteurs MHEG-3 l'accès à ces fonctions externes. Une telle convention, appelée spécification de mappage de plate-forme, dépend de la plate-forme d'exécution.

Les spécifications de mappage de plate-forme conformes aux dispositions de la présente Recommandation doivent être enregistrées pour assurer l'interopérabilité des services de l'environnement d'exécution avec tout moteur MHEG-3 de cette plate-forme. Si une spécification de mappage de plate-forme existe pour la plate-forme, les moteurs MHEG-3 devront être conformes à cette spécification de mappage de plate-forme pour accéder aux services de l'environnement d'exécution.

Les implémentations de moteur MHEG-3 contiendront dans leur document de conformité la ou les spécifications de mappage de plate-forme auxquelles ils sont conformes.

NOTE – Lorsqu'un logiciel existant non conforme à la spécification de mappage de plate-forme a besoin d'accéder à des objets scripts, il peut être encapsulé dans une interface traduisant ses propres conventions d'interfaces dans celles de la spécification de mappage de plate-forme.

10.2 Déclaration des interfaces IDL

L'interface des logiciels externes devant être utilisés par un script échangé peut contenir

– des déclarations d'opérations;

– des déclarations d'exceptions;

– des déclarations de types.

Les types seront déclarés dans la déclaration de type du script échangé.

Les opérations et les exceptions seront déclarées dans la déclaration de bloc du script échangé. On affectera un identificateur de bloc à cette déclaration de bloc composé:

– du nom du bloc;

– d'un ensemble de description de services;

– d'un ensemble de descriptions d'exceptions.

On affectera un identificateur de fonction aux descriptions de service composé:

– du nom de l'opération;

– de la signature de la fonction (type, mode de passage de chaque paramètre et type de la valeur de retour).

On affectera un identificateur de message aux descriptions d'exception composé:

– du nom de l'exception;

– de la signature de l'exception (type de chaque paramètre).

Les identificateurs (identificateurs de fonction, de type et de bloc) sont utilisés par les scripts SIR MHEG pour référencer des types et des fonctions, On peut construire un identificateur de fonction concernant une opération externe à partir d'un identificateur de bloc et l'index de la déclaration de service dans ce bloc, tandis qu'on peut construire celui d'une exception externe à partir de l'identificateur de bloc et l'index de la déclaration d'exception dans ce bloc.

L'interpréteur de script utilisera les noms (noms de bloc, d'opération et d'exception) pour effectuer des liens avec l'implémentation réelle du logiciel externe.

Une déclaration de bloc SIR-MHEG se situe au même niveau d'abstraction que celui d'une spécification IDL. La présente Recommandation définit les règles de mappage entre une spécification IDL et une déclaration de bloc. Le paragraphe 14 spécifie

– le mappage entre une description de type de donnée IDL et celle d'un type de donnée SIR MHEG;

– le mappage entre une description d'opération IDL et celle d'un service SIR MHEG;

– le mappage entre une description d'exception IDL et celle d'une exception SIR MHEG.

10.3 Invocation d'opérations externes dans un programme SIR MHEG

A partir d'un programme SIR MHEG, on invoquera une déclaration de bloc décrivant un service de la manière suivante:

– les variables des types attendus correspondant à la valeur de retour (si elle existe) et à chaque paramètre seront déclarées dans le script échangé (sauf la référence à objet de l'objet d'origine qui sera implicite);

– le programme affectera ces variables qui correspondront à des paramètres d'entrée ou d'entrée/sortie.

– le programme empilera les identificateurs de donnée des variables selon un ordre de type droite-gauche [l'identificateur de la variable correspondant à la valeur de retour est empilé en premier suivi des paramètres effectifs pour terminer par la référence à objet (paramètre implicite) de la destination];

– Le programme utilisera l'instruction **appel externe (XCALL)** pour invoquer l'opération avec l'identificateur de fonction comme opérande;

– le programme exploitera les résultats de la fonction en utilisant les variables correspondant à la valeur de retour (paramètres **out** et **inout**).

10.4 Manipulation d'exceptions externes dans un programme SIR MHEG

Un programme SIR MHEG manipulera une exception décrite dans une déclaration de bloc de la manière suivante:

– les variables des types attendus correspondant à chaque membre seront déclarées dans le script échangé (à l'exception de la référence à objet de l'objet d'origine qui sera implicite);

– une routine, dont les paramètres correspondent aux membres d'exception, sera déclarée dans la partie déclaration de la routine du script échangé;

– Le mappage entre les identificateurs de cette routine de manipulation et l'exception sera déclarée dans la partie déclaration de filet du script échangé.

10.5 Invocation d'opérations externes par un moteur MHEG-3

Lorsqu'un script échangé exprimera l'invocation d'une opération selon la description faite au 10.3, l'interpréteur de script se comportera selon la description correspondant à la sémantique de l'instruction **XCALL** du paragraphe 13. Il effectuera l'interprétation des mécanismes décrits au 10.3 en les traduisant en des mécanismes d'accès à l'environnement d'exécution selon la définition des spécifications de mappage de la plate-forme.

NOTE – Par exemple, un moteur MHEG-3 peut traduire un identificateur de variable empilé comme un paramètre de service en soit une valeur ou une adresse mémoire réelle qui sera ensuite passée au logiciel externe fournissant le service.

10.6 Manipulation des exceptions externes par un moteur MHEG-3

L'apparition d'une exception déclenchée par un service externe a pour effet de transmettre un message au moteur MHEG-3 selon les mécanismes d'accès à l'environnement d'exécution définis par les spécifications de mappage de plate-forme.

L'interpréteur de script se comportera alors selon la description faite au 9.5.1.2.

10.7 Spécifications de mappage de plate-forme

Une spécification de mappage de plate-forme contiendra tout ce qui suit:

– la description de la plate-forme pour laquelle s'applique la spécification;

– la procédure **package availability** que les moteurs MHEG-3 utiliseront pour vérifier la disponibilité d'un bloc donné dans l'environnement d'exécution;

– la procédure **package load** que les moteurs MHEG-3 utiliseront pour rendre les opérations d'un bloc donné accessibles à un *rt-script*;

– la procédure **package unload** que les moteurs MHEG-3 utiliseront pour décharger un bloc;

– la procédure **operation invocation** que les moteurs MHEG-3 utiliseront pour invoquer une opération donnée;

– les règles de codage de données (**data encoding** rules**)** que les moteurs MHEG-3 utiliseront pour coder les valeurs des paramètres **in** ou **inout** d'une opération et pour décoder les valeurs des paramètres **out** ou **inout** d'une opération ou des membres d'exception;

– les procédures **parameter passing** que les moteurs MHEG-3 utiliseront pour passer à une opération les paramètres **in, inout** et **out**;

– la procédure **return value retrieval** que les moteurs MHEG-3 utiliseront pour récupérer la valeur de retour d'une opération;

– la procédure **exception retrieval** que les moteurs MHEG-3 utiliseront pour récupérer les exceptions soulevées par une opération.

L'Annexe D définit les contenus d'une spécification de mappage de plate-forme.

11 Dispositions pour la manipulation d'objets MHEG

Le présent paragraphe décrit les mécanismes définis par la présente Recommandation pour rendre possible la manipulation d'objets MHEG par des *rt-scripts*.

11.1 Invocation d'action MHEG

On utilise la SIR MHEG pour exprimer des invocations d'actions selon les définitions données par l'API MHEG.

L'API MHEG est définie au moyen d'IDL. Le mappage à partir d'une définition IDL vers des déclarations de bloc et de type de la SIR MHEG est définie dans le paragraphe 14. Cependant, le bloc API MHEG est considéré comme un bloc prédéfini. Ainsi on n'inclura pas sa déclaration de manière explicite dans les script échangés. Le mécanisme de mappage est identique à celui de la déclaration de fonction externe décrite dans le paragraphe 10, à ceci près que les types et opérations IDL définies par l'API MHEG ne seront pas déclarées comme des parties de code SIR MHEG mais plutôt traitées comme des types et fonctions externes prédéfinis.

Le mécanisme utilisé pour invoquer une action MHEG est identique à l'invocation d'un service fourni par l'environnement d'exécution. On utilise une instruction **XCALL**. On référence les types définis dans le bloc API MHEG en utilisant un identificateur de type prédéfini. On référence les fonctions décrites dans l'API MHEG en utilisant un identificateur de fonction prédéfinie.

11.1.1 Envoi de messages à d'autres scripts

Le bloc API MHEG-3 est considéré comme un bloc prédéfini. Dans un script échangé, les messages peuvent être efficacement dirigés vers d'autres scripts en utilisant les fonctions prédéfinies mappant les opérations de l'API MHEG-3. Deux *rt-scripts* peuvent ainsi se transmettre des paramètres et des routines d'appel.

NOTE – Ceci peut être utilisé pour implémenter les concepts de bibliothèque ou d'utilitaire de script. Ceci peut aussi être utilisé pour synchroniser des *rt-scripts*.

11.1.2 Echange d'information avec des objets MHEG

On peut exprimer un échange d'information entre un *rt-script* et d'autres entités MHEG (y compris d'autres *rt-scripts*) en utilisant les opérations de l'API MHEG correspondant aux actions MHEG "set data" et "get data". On peut utiliser les objets MHEG **content** comprenant des valeurs génériques afin de constituer une zone de mémoire partagée entre des objets MHEG.

On peut traduire une attente de signal en provenance d'un autre objet MHEG par une boucle incluant un appel à l'opération de l'API MHEG correspondant à l'action MHEG "get data" jusqu'à la récupération de la valeur attendue.

On peut traduire la génération d'un signal par un appel à l'opération de l'API MHEG correspondant à l'action MHEG "set data".

NOTE – On recommande d'utiliser le mécanisme décrit au 11.1.1 pour ce qui concerne l'échange d'information entre des *rt-scripts*.

11.2 Réception de messages MHEG

On utilise la SIR MHEG pour exprimer la manipulation de messages résultant d'actions MHEG. Ces messages peuvent être l'un des suivants:

– opérations **run** de l'API MHEG-3;

– exceptions de l'API MHEG.

11.2.1 Opération run de l'API MHEG-3

Les actions MHEG "run" et "set parameters" pouvant être dirigées vers un *rt-script* devraient aboutir à des opérations **run** et **setParameter** de l'API MHEG-3. L'invocation de l'opération **run** a pour résultat l'insertion d'un message dans la file d'attente des messages du *rt-script* avec:

– comme identificateur de message, un identificateur de message prédéfini mappé sur l'identificateur de routine de la routine destination;

– comme membres, les paramètres précédemment affectés par l'opération **setParameter**.

11.2.2 Exceptions de l'API MHEG

Les exceptions de l'API MHEG sont considérées comme des messages envoyés à l'interpréteur de script comme le résultat de l'invocation d'une opération de l'API MHEG-3. Ces exceptions possèdent des identificateurs de message prédéfini. L'interpréteur de script traitera ces messages comme s'il devait traiter une exception provenant de l'environnement d'exécution selon la description faite au 9.5.1.2.

12 Déclarations de la SIR MHEG

Le présent paragraphe définit la structure des scripts échangés. Le présent paragraphe spécifie aussi la manière dont la machine virtuelle effectue l'analyse syntaxique d'un script échangé.

On utilise les conventions de notation suivantes:

– les non-terminaux sont écrits en texte normal;

– Les types terminaux sont écrits en MAJUSCULES;

– Les valeurs énumérées sont encadrées entre 'quotes';

– ":=" indique une définition;

– "|" indique un choix dans une production;

– "*" indique zéro ou plusieurs occurrences du type précédent;

– "+" indique une ou plusieurs occurrences du type précédent;
– "?" indique zéro ou une occurrence du type précédent (type optionnel).

NOTE – La grammaire complète des scripts échangés est décrite dans l'Appendice I.

Un script échangé comprendra:
– une séquence de déclarations de types;
– une séquence de déclarations de constantes;
– une séquence de déclarations de variables globales;
– une séquence de déclarations de blocs;
– une séquence de déclarations de filets de messages;
– une séquence de déclarations de routines.

InterchangedScript ::= TypeDeclaration*
ConstantDeclaration*
VariableDeclaration*
PackageDeclaration*
HandlerDeclaration*
RoutineDeclaration*

12.1 Déclaration de type

On utilise les déclarations de types pour décrire les types du script échangé.

Une déclaration de type comprend:
– un identificateur de type (optionnel);
– une description de type.

TypeDeclaration ::= TypeIdentifier?
TypeDescription

12.1.1 Identificateur de type

On utilise les identificateurs de types pour référencer la description d'un type d'un bout à l'autre du script échangé.

L'identificateur de type sera un entier positif compris dans l'intervalle alloué aux types déclarés. Il correspondra au nombre maximal de types auquel on ajoutera l'index (à partir de zéro) de sa déclaration dans la partie déclarations de types.

Lorsque l'identificateur de type n'est pas fourni, il doit être calculé par l'analyseur syntaxique de script.

TypeIdentifier ::= INTEGER

12.1.2 Description de type

Les descriptions de types décrivent la structure d'un type déclaré.

Une description de type sera l'une des suivantes:
– description de chaîne;
– description de séquence;
– description de tableau;
– description de structure;

– description d'union.

```
TypeDescription     ::=  SequenceDescription
                    |    StringDescription
                    |    ArrayDescription
                    |    StructureDescription
                    |    UnionDescription
```

12.1.2.1 Description de chaîne

Une description de chaîne sera composée d'un entier (optionnel)

```
StringDescription   ::=  INTEGER?          // String (max) size
```

Cet entier représente la taille maximale de la chaîne; s'il n'est pas fourni, la chaîne sera dite non liée.

12.1.2.2 Description de séquence

Une description de séquence sera composée:

– d'un entier (optionnel);

– d'un identificateur de type.

```
SequenceDescription     ::=  INTEGER?          // Sequence (max) size
                             TypeIdentifier
```

Cet entier représente la taille maximale de la séquence; s'il n'est pas fourni, la séquence sera dite non liée.

L'identificateur de type représente le type d'élément de la séquence.

12.1.2.3 Description de tableau

Une description de tableau sera composée:

– d'un entier;

– d'un identificateur de type.

```
ArrayDescription        ::=  INTEGER           // Array size
                             TypeIdentifier
```

Cet entier représente le type d'élément du tableau.

L'identificateur de type représente le type d'élément du tableau.

12.1.2.4 Description de structure

Une description de structure sera composée d'une séquence d'identificateurs de types.

```
StructureDescription    ::=  TypeIdentifier+
```

Chaque identificateur de type représente le type d'un des champs de la structure.

12.1.2.5 Description d'union

Une description d'union sera composée d'une séquence d'un ou plusieurs identificateurs de types.

```
UnionDescription        ::=  TypeIdentifier+
```

Chaque identificateur de type représente le type d'un des choix de l'union.

12.2 Déclaration de constante

On utilise les déclarations de constantes pour décrire les types et les valeurs des constantes du script échangé.

Une déclaration de constante sera composée:

– d'un identificateur de donnée (optionnel);

– d'un identificateur de type;

– d'une valeur de constante.

```
ConstantDeclaration     ::=  DataIdentifier?
                             TypeIdentifier
                             ConstantValue
```

12.2.1 Identificateur de donnée

On utilise des identificateurs de données pour référencer des données d'un bout à l'autre du script échangé.

L'identificateur de donnée sera un nombre positif compris dans l'intervalle alloué aux constantes. Il correspondra à l'index (à partir de 0) de sa déclaration dans la partie déclarations de constantes.

Lorsque les identificateurs de données ne sont pas fournis, il sont alors calculés par l'analyseur syntaxique de script.

```
DataIdentifier          ::=  INTEGER
```

12.2.2 Identificateur de type

L'identificateur de type représente le type auquel appartient la valeur de la constante.

12.2.3 Valeur de constante

La valeur de constante représente la valeur correspondant à la constante d'un bout à l'autre du script.

Si le type de la constante est un type primitif ou **string**, la valeur de la constante sera composée d'une valeur immédiate exprimée dans ce type.

Si le type de la constante est un type **sequence**, la valeur de la constante sera composée d'une séquence de valeurs constantes dont la longueur est inférieure ou égale à la taille du type **sequence** et dont le type est le type d'élément de la description de séquence.

Si le type de la constante est un type **array**, la valeur de la constante sera composée d'une séquence de valeurs constantes dont la longueur est égale à la taille du type **array** et dont le type est le type d'élément de la description de tableau.

Si le type de la constante est un type **structure**, la valeur de la constante sera composée d'une séquence de valeurs constantes dont la longueur est égale aux nombre d'éléments dans le type **structure** chacune de ces valeurs sera d'un type identique à celui du type de l'élément correspondant dans la description de structure.

Si le type de la constante est un type **union**, la valeur de la constante sera composée d'un entier représentant l'index (à partir de 0) du choix dans le type **union** et d'une valeur constante dont le type est le type de l'élément de rang correspondant dans la description de l'union.

```
ConstantValue           ::=  BOOLEAN
                        |    OCTET
                        |    INTEGER    // all numeric types
                        |    REAL               // float or double
```

```
                | STRING      // character or string
                | DataIdentifier
                | ConstantValue*    // sequence, array or structure
                | UnionValue

UnionValue      ::= INTEGER   // Tag index
                    ConstantValue
```

12.3 Déclaration de variable globale

On utilise les déclarations de variables globales pour décrire les types et les valeurs initiales des variables globales du script échangé.

Une déclaration de variable globale sera composée:

– d'un identificateur de donnée (optionnel);

– d'un identificateur de type;

– d'une référence à constante (optionnel).

```
VariableDeclaration     ::= DataIdentifier?
                            TypeIdentifier
                            ConstantReference? // Initial value
```

12.3.1 Identificateur de donnée

On utilise des identificateurs de données pour référencer des données d'un bout à l'autre du script échangé.

L'identificateur de donnée sera un nombre positif compris dans l'intervalle alloué aux variables globales. Il correspondra aux nombre maximal de constantes incrémenté de l'index (à partir de 0) de la déclaration dans la partie déclarations de variables globales.

Lorsque les identificateurs de données ne sont pas fournis, il sont alors calculés par l'analyseur syntaxique de script.

12.3.2 Identificateur de type

L'identificateur de type représente le type auquel appartient la valeur de la variable globale.

12.3.3 Référence à constante

La référence à constante représente la valeur initiale d'une variable globale.

La référence à constante sera l'un des éléments suivants:

– un identificateur de donnée référençant une constante;

– une valeur constante selon la description du 12.2.3.

Dans tous les cas, la valeur référencée par cette référence à constante sera du type de la variable globale.

Si la référence à constante n'est pas fournie, l'interpréteur de script affectera à la valeur globale une valeur par défaut si son type le permet. Sinon, elle restera indéfinie jusqu'à son affectation par une instruction.

```
ConstantReference   ::= DataIdentifier
                    | ConstantValue
```

12.4 Déclaration de bloc

On utilise des déclarations de bloc pour décrire les services et exceptions externes utilisées par le script échangé.

Une déclaration de bloc comprendra:

– un identificateur de bloc (optionnel);

– une chaîne contenant le nom du bloc;

– une séquence de descriptions de services;

– une séquence de descriptions d'exceptions.

```
PackageDeclaration      ::=     PackageIdentifier?
                                VisibleString       // Package name
                                ServiceDescription*
                                ExceptionDescription*
```

12.4.1 Identificateur de bloc

On utilise des identificateurs de blocs pour référencer des blocs d'un bout à l'autre du script échangé.

L'identificateur de bloc sera un nombre positif compris dans l'intervalle alloué aux blocs. Il correspondra à l'index (à partir de 0) de sa déclaration dans la partie déclarations de blocs.

Lorsque les identificateurs de blocs ne sont pas fournis, il sont alors calculés par l'analyseur syntaxique de script.

```
PackageIdentifier       ::=     INTEGER
```

12.4.2 Nom

L'interpréteur de script utilise un nom de bloc pour y accéder à l'intérieur de l'environnement d'exécution, conformément à la procédure **package availability** décrite par la spécification de mappage de plate-forme.

12.4.3 Description de service

Les descriptions de services décrivent des prototypes de fonctions externes.

Une description de service comprendra:

– un identificateur de fonction (optionnel);

– une chaîne contenant le nom d'opération;

– un mode d'appel (optionnel);

– un identificateur de type (optionnel);

– une séquence de descriptions de paramètres.

```
ServiceDescription      ::=     FunctionIdentifier?
                                VisibleString?      // IDL global name
                                CallingMode?
                                TypeIdentifier?     // return value
                                ServiceParameterDescription*
```

12.4.3.1 Identificateur de fonction

On utilise des identificateurs de fonctions pour référencer des fonctions d'un bout à l'autre du script échangé.

L'identificateur de fonction sera un nombre positif compris dans l'intervalle alloué aux services. Il correspondra au nombre maximal de routines ajouté du nombre maximal de fonctions prédéfinies et de l'identificateur de bloc multiplié par 256, le tout incrémenté de l'index (à partir de 0) du service dans la déclaration de bloc.

Lorsque l'identificateur de fonction n'est pas fourni, il est alors calculé par l'analyseur syntaxique de script.

FunctionIdentifier ::= INTEGER

12.4.3.2 Nom

L'interpréteur de script utilise le nom d'opération pour accéder à l'opération à l'intérieur de l'environnement d'exécution, conformément à la procédure **operation invocation** décrite par la spécification de mappage de plate-forme.

12.4.3.3 Mode d'appel

Le mode d'appel représente la manière d'invoquer l'opération.

Le mode d'appel sera soit 'synchrone', soit 'asynchrone'.

Si la valeur n'est pas spécifiée, le mode d'appel sera 'synchrone'.

CallingMode ::= 'SYNCHRONOUS' | 'ASYNCHRONOUS'

12.4.3.4 Identificateur de type

L'identificateur de type représente le type de la valeur de retour du service.

L'identificateur de type sera interprété avec un type **void** lorsqu'il n'est pas spécifié; la fonction n'aura donc pas de valeur de retour.

Si le mode d'appel de l'opération est 'asynchrone', l'identificateur de type sera soit de type "void" ou non spécifié.

12.4.3.5 Description de paramètre

On utilise les descriptions de paramètres pour spécifier le type et le mode de passage des paramètres du service.

Une description de paramètre sera composée

– d'un mode de passage;

– d'un identificateur de type.

ServiceParameterDescription ::= ServicePassingMode?
TypeIdentifier

12.4.3.5.1 Mode de passage

Le mode de passage indique si la valeur du paramètre au moment de l'invocation du service est utilisée par le service (paramètre d'entrée) et si ce paramètre est modifié par le service en vue d'une utilisation par celui qui appelle le service (paramètre de sortie).

Le mode de passage sera soit de type **in**, **inout** ou **out**.

Si le mode de passage n'est pas spécifié, le paramètre sera interprété comme étant de type **in**.

NOTE – Le paramètre de référence à objet est implicite, aussi ne devrait-on pas le spécifier dans la partie déclaration. Il sera traité comme un paramètre de type **in**.

Si le mode d'appel de l'opération est 'asynchrone', le mode de passage sera soit de type **in** ou non spécifié.

ServicePassingMode ::= 'IN' | 'OUT' | 'INOUT'

12.4.3.5.2 Identificateur de type

L'identificateur de type représente le type du paramètre du service considéré.

12.4.4 Description d'exception

Les descriptions d'exceptions décrivent des prototypes d'exceptions pouvant apparaître pendant l'exécution de fonctions externes.

Une description d'exception sera composée:

– d'un identificateur de message (optionnel);

– d'une chaîne représentant le nom de l'exception;

– d'une séquence d'identificateurs de types représentant les membres de l'exception.

```
ExceptionDescription ::= MessageIdentifier?
                         VisibleString?     //IDL exception global name
                         TypeIdentifier*    //Parameter types
```

12.4.4.1 Identificateur de message

On utilise les identificateurs de messages pour référencer des messages tout au long du script échangé.

L'identificateur de message sera un entier positif compris dans l'intervalle alloué aux exceptions. Il correspondra au nombre maximal de messages prédéfinis, ajouté à l'identificateur de bloc multiplié par 256 et incrémenté de l'index (à partir de 0) de l'exception dans la déclaration de bloc.

S'il n'est pas fourni, l'identificateur de message sera calculé par l'analyseur syntaxique de script.

MessageIdentifier ::= INTEGER

12.4.4.2 Nom

L'interpréteur de script utilise un nom d'exception pour récupérer l'exception dans l'environnement d'exécution, conformément à la procédure **exception retrieval** décrite par la spécification de mappage de plate-forme.

12.4.4.3 Description de paramètre

Chaque paramètre du message correspond à un membre de l'exception et est décrit par son identificateur de type.

12.5 Déclaration de filet

On utilise une déclaration de filet pour associer un message à la fonction qui le manipule.

Une déclaration de filet sera composée:

– d'un identificateur de message;

– d'un identificateur de fonction.

```
HandlerDeclaration ::= MessageIdentifier
                       FunctionIdentifier
```

12.5.1 Identificateur de message

L'identificateur de message indique quel message est à manipuler.

L'identificateur de message sera un entier positif compris dans tout l'intervalle alloué aux messages, représentant soit un message prédéfini, soit une exception.

12.5.2 Identificateur de fonction

L'identificateur de fonction indique la fonction à déclencher lorsque le message est ôté de la file d'attente des messages.

L'identificateur de fonction sera un entier positif compris dans tout l'intervalle alloué aux fonctions, représentant soit une fonction prédéfinie, soit un service.

La description des types de paramètres formels de la fonction sera identique à celle utilisée pour les messages, de sorte que la fonction puisse être appelée avec en paramètres les paramètres effectifs du message. Si les signatures ne correspondent pas, le filet sera rejeté par l'analyseur syntaxique de script.

12.6 Déclaration de routine

On utilise les déclarations de routines pour décrire la structure et le code programme des fonctions internes du script échangé.

Une déclaration de routine sera composée:

– d'un identificateur de fonction (optionnel);

– d'un identificateur de type (optionnel);

– d'une séquence de descriptions de paramètres;

– d'une séquence de déclarations de variables locales;

– d'un code programme SIR MHEG.

```
RoutineDeclaration      ::=     FunctionIdentifier?
                                TypeIdentifier?             // for return value
                                RoutineParameterDescription*
                                LocalVariableDeclaration*
                                OCTET STRING                // program code
```

12.6.1 Identificateur de fonction

L'identificateur de fonction sera un entier positif compris dans tout l'intervalle alloué aux routines. Il correspondra à l'index (à partir de 0) de la routine dans la partie déclarations de routines.

S'il n'est pas fourni, l'identificateur de fonction sera calculé par l'analyseur syntaxique de script.

12.6.2 Identificateur de type

L'identificateur de type représente le type de la valeur de retour de la routine.

S'il n'est pas spécifié, l'identificateur de type sera interprété comme un type **void**; la fonction n'aura donc pas de valeur de retour.

12.6.3 Description de paramètre

On utilise les descriptions de paramètres pour spécifier le type et le mode de passage des paramètres de routine.

Une description de paramètre sera composée

– d’un mode de passage (optionnel);

– d’un identificateur de type.

RoutineParameterDescription ::= RoutinePassingMode?
TypeIdentifier

12.6.3.1 Mode de passage

Le mode de passage indique si le paramètre sera passé à la routine par valeur (paramètre d’entrée) ou par référence à la variable contenant la valeur (paramètre d’entrée et de sortie)

Le mode de passage sera de type ‘value’ ou ‘reference’.

RoutinePassingMode ::= ’VALUE’ | ’REFERENCE’

12.6.3.2 Identificateur de type

L’identificateur de type représente le type du paramètre de la routine considérée.

12.6.4 Déclaration de variable locale

Les déclarations de variables locales servent à décrire les types et les valeurs initiales des variables dont la portée est limitée à une seule exécution de routine.

Une déclaration de variable locale aura la même structure qu’une déclaration de variable globale (voir 12.3). Elle sera composée:

– d’un identificateur de donnée (optionnel);

– d’un identificateur de type;

– d’une référence à constante (optionnel).

12.6.4.1 Identificateur de donnée

L’identificateur de donnée sera un nombre positif compris dans l’intervalle alloué aux variables locales. Il correspondra au nombre maximal de constantes, ajouté du nombre maximal de variables globales incrémenté de l’index (à partir de 0) de sa déclaration dans les déclarations de variables locales de la routine et incrémenté du nombre de paramètres formels de la routine.

Si l’identificateur de donnée n’est pas fourni, il sera calculé par l’analyseur syntaxique de script.

12.6.4.2 Identificateur de type

L’identificateur de type représente le type auquel appartient la valeur de la variable locale.

12.6.4.3 Référence à constante

La référence à constante représente la valeur initiale de la variable locale.

La référence à constante sera:

– soit un identificateur de donnée référençant une constante;

– soit une valeur de constante conforme à la définition donnée au 12.2.3.

Dans tous les cas, la valeur référencée par la référence à constante sera du type de la variable locale.

Si la référence à constante n’est pas fournie, l’interpréteur de script affectera à la variable locale une valeur par défaut si le type le permet. Sinon, elle restera indéfinie jusqu’à son affectation par une instruction.

12.6.5 Code programme

Le code programme est composé d'une séquence d'instructions dans la routine, qui seront exécutées par l'interpréteur de script lors d'un appel à la routine. Le paragraphe 13 décrit la syntaxe et la sémantique des instructions SIR MHEG.

La dernière instruction d'une routine sera une instruction **RET**.

13 Instructions de la SIR MHEG

Le présent paragraphe définit la sémantique des instructions SIR MHEG.

13.1 Méthodologie de présentation

Chaque instruction est définie dans un sous-paragraphe correspondant par un ensemble d'entrées comme indiqué ci-dessous:

Description succincte:	brève description de la sémantique de l'instruction.
synopsis:	**Mnemonique Opérande1 ... OpérandeN**
opérandes:	description des types et de la sémantique de chaque opérande accolé à l'instruction (s'il en existe).
pile:	synopsis visuel de l'effet de l'instruction sur la pile des paramètres; par exemple: **..., Paramètre1, Paramètre2 ⇨ ..., Résultat**
types:	liste des types de paramètres s'appliquant à l'instruction (si l'instruction est un modèle).
paramètres:	description de la sémantique de chaque élément empilé sur ou dépilé de la pile de paramètres ou, sur lequel l'instruction à un effet (en cas de besoin).
effet:	spécification textuelle de la sémantique d'interprétation de l'instruction.
spécification formelle:	spécification formelle de la sémantique d'interprétation de l'instruction à l'aide de la notation décrite dans le présent sous-paragraphe.
erreurs:	liste des erreurs pouvant apparaître lors de l'exécution de l'instruction.

13.1.1 Conditions d'erreur

Conformément à la description de la spécification formelle, la sémantique d'une instruction sera appliquée uniquement si ses opérandes sont valides. Dans le cas contraire, une exception **InstructionExecutionError** apparaîtra et le registre d'erreur sera mis à la valeur **InvalidOperand**. On ne spécifie pas le résultat de l'exécution.

Si la pile de paramètres ne contient pas suffisamment de paramètres lorsqu'on y accède à l'aide des primitives **PS.pop** ou a **PS[SP]**, une exception **InstructionExecutionError** apparaîtra et le registre d'erreur sera mis à la valeur **StackUnderflow**. On ne spécifie pas l'état de la pile des paramètres.

Si le résultat d'une opération arithmétique est dans un intervalle dépassant celui du type considéré, les opérations arithmétiques feront apparaître une exception **InstructionExecutionError** et le registre d'erreur sera mis à la valeur **ArithmeticOverflow** ou **DivisionByZero** selon le type d'erreur.

Si un identificateur fait référence à une entité non valide (type, donnée, fonction, message, bloc), lorsqu'on accède à sa valeur dans la table correspondante à l'aide de **DT[i]**, une exception **InstructionExecutionError** apparaîtra et le registre d'erreur sera mis à la valeur **InvalidIdentifier**.

Si IP contient un pointeur non valide, une exception **InstructionExecutionError** apparaîtra et le registre d'erreur sera mis à la valeur **JumpOutOfRange**.

Lors de l'affectation d'une variable dynamique, si l'affectation est impossible à cause d'un manque d'espace mémoire ou d'identificateur de données, alors la primitive **new**() fera apparaître une exception **InstructionExecutionError** et mettra le registre d'erreur à la valeur **AllocationFailed.**

Le déclenchement des autres conditions d'erreur est expliqué au 13.3. Les valeurs de code d'erreur sont définies dans l'Annexe C.

13.1.2 Spécification formelle

L'entrée " spécification formelle " d'une description d'entrée propose une notation formelle et précise de l'effet que l'unité d'exécution d'instructions produira lors de l'interprétation de l'instruction; cependant, cette spécification étant exprimée sous forme de séquence d'opérations, il peut y avoir d'autres méthodes conduisant au même résultat. Aussi n'obligera–t'on pas une unité d'exécution d'instructions à traiter l'instruction comme indiqué à partir du moment où l'effet produit est le même.

Les cas d'erreur décrits au 13.1.1 sont implicites et ne sont pas exprimés à l'aide de la spécification formelle. Les autres cas d'erreur sont mentionnés de manière explicite.

On utilise une syntaxe de type C pour spécifier formellement la sémantique d'une instruction. Celle-ci utilise les notations et les concepts définis dans les paragraphes 8 et 9 avec en plus les notations suivantes:

– notation de table de données (DT, *data table*) (voir 13.1.3);
– notation d'instruction générique (voir 13.1.4);
– primitives (voir 13.1.5).

13.1.3 Notation de table de données (data table)

La notation **DT(i)**, dans laquelle **i** est un identificateur de donnée, correspond à:

– l'entrée dont la clé est **i** dans la table des constantes, si **i** est l'identificateur de donnée d'une constante;
– l'entrée dont la clé est **i** dans la table des variables globales, si **i** est l'identificateur de donnée d'une variable globale;
– l'entrée dont la clé est **i** dans la table des variables locales, si **i** est l'identificateur de donnée d'une variable locale;
– la variable dynamique correspondant à **i**, si **i** est l'identificateur de donnée d'une variable dynamique.

Cette macro peut être exprimée comme suit:

```
#define DT(i) (i < 4096) ? CT[i] : VT[i])
```

13.1.4 Notation d'instruction générique

Un nombre d'instructions logiques et arithmétiques opèrent sur des valeurs d'un type donné et produisent un résultat du même type. On utilise la notation <**T**> pour exprimer une instruction générique. <**Mnemonic**>_<**T**> représente alors toutes les instructions <**Mnemonic**> avec <**T**> devant être remplacé par la lettre du type de tout type de primitive à laquelle l'instruction peut s'appliquer. Cette lettre est donnée dans l'entrée "Types" de la description d'instruction.

NOTE – Les opérations portant sur des types mixtes devraient être effectuées en insérant de manière explicite des instructions de conversion de type dans la séquence d'instructions.

13.1.5 Primitives

On utilise les notations de primitives suivantes dans la spécification formelle des instructions:

– **DID new(tid)**: alloue une variable dynamique avec un type identifié dans **tid**;

– **void raise(exc)**: soulève une exception **InstructionExecutionError** et affecte le code d'erreur **exc**;

– **void delete(did)**: libère la variable dynamique identifiée par **did**;

– **int sizeof(tid)**: renvoie la taille des valeurs du type identifié par **tid**, exprimée dans les mêmes unités que celles adressées par le pointeur PS;

– **type(<T>)**: macro à remplacer par le nom de type C.

13.2 Classification des instructions de la SIR MHEG

Les instructions de la SIR MHEG peuvent être regroupées en catégories selon leur effet sur le contrôle de flux, les tables de variables ou la pile de paramètres, et selon les types de paramètres de pile qu'elles acceptent:

a) instructions ayant un effet sur le contrôle de flux:
   1) instructions de saut inconditionnel: **JMP, LJMP**;
   2) instructions de saut conditionnel: **JT, JF, LJT, LJF**;
   3) invocations de fonction: **CALL, XCALL**;
   4) instructions de contrôle de flux diverses: **RET, YIELD**.

b) instructions n'ayant pas d'effet sur le contrôle de flux, mais affectant la valeur des variables:
   1) modificateurs de variables complexes: **SET, SETC**;
   2) opérateurs arithmétiques sur les variables: **INC, DEC**;
   3) instructions d'empilement: **POPR, POP, POPC**;
   4) instructions de gestion mémoire: **ALLOC, FREE**.

c) instructions n'ayant pas d'effet sur le contrôle de flux ou les variables, mais affectant la pile de paramètres:
   1) instructions de conversion: **CVT**;
   2) opérateurs arithmétiques: **ADD, SUB, MUL, DIV, REM, NEG**;
   3) opérateurs logiques: **AND, OR, XOR, NOT**;
   4) opérateurs de décalage logique: **SHIFT**;
   5) opérateurs de comparaison: **EQ, GT, LT, EQR**;
   6) extracteurs de données complexes: **GET, GETC**;
   7) instructions diverses de manipulation de pile: **PUSHI, PUSHR, PUSH, DUP, GETOR**.

d) instruction n'ayant aucun effet: **NOP**.

NOTE – La plupart des instructions opèrent seulement sur des valeurs de types primitifs. Seules les instructions suivantes sont utilisées pour manipuler des types construits: **EQR, GET, GETC, SET, SETC, ALLOC, FREE, CALL, XCALL**.

L'effet des instructions est résumé dans le Tableau 3. Les opérations sont listées selon un ordre canonique, par ordre ascendant de code opérateur. Certains mnémoniques correspondent à des instructions génériques et ont donc des suffixes typés.

Tableau 3/T.173 – Synopsis et effets des instructions de la SIR MHEG

Mnémonique	Ref.	Code opérateur (hexa)	Taille du code op	Type du code op	Types des paramètres	Effet sur PS	Effet sur VT	Effet sur le contrôle de flux
NOP	13.3.1	00	0					
YIELD	13.3.2	02	0					x
RET	13.3.3	03	0			0|1 ⇨ 0|1		x
FREE	13.3.4	08	0			1 ⇨ 0	x	
NOT_<T>	13.3.5	10-13	0		BOWU	1 ⇨ 1		
OR_<T>	13.3.6	14-17	0		BOWU	2 ⇨ 1		
XOR_<T>	13.3.7	18-1B	0		BOWU	2 ⇨ 1		
AND_<T>	13.3.8	1C-1F	0		BOWU	2 ⇨ 1		
EQR	13.3.9	20	0			2 ⇨ 1		
EQ_<T>	13.3.10	21-2B	0		OSLWUFDBCIR	2 ⇨ 1		
LT_<T>	13.3.11	30-37	0		COSLWUFD	2 ⇨ 1		
GT_<T>	13.3.12	38-3F	0		COSLWUFD	2 ⇨ 1		
ADD_<T>	13.3.13	40-47	0		OSLWUFD	2 ⇨ 1		
SUB_<T>	13.3.14	48-4F	0		OSLWUFD	2 ⇨ 1		
MUL_<T>	13.3.15	50-57	0		OSLWUFD	2 ⇨ 1		
DIV_<T>	13.3.16	58-5F	0		OSLWUFD	2 ⇨ 1		
NEG_<T>	13.3.17	62-67	0		SLFD	1 ⇨ 1		
REM_<T>	13.3.18	79-7D	0		OSLWU	2 ⇨ 1		
DUP_<T>	13.3.19	81-8B	0		OSLWUFDBCIR	1 ⇨ 2		
CVT_<TT>	13.3.20	94-BE	0		OSLWUFDBC	1 ⇨ 1		
JT	13.3.21	C0	1	offset		1 ⇨ 0		x
JF	13.3.22	C1	1	offset		1 ⇨ 0		x
JMP	13.3.23	C2	1	offset				x
SHIFT_<T>	13.3.24	C5-C7	1	offset	OWU	1 ⇨ 1		
GETOR	13.3.25	C9	1	PID		0 ⇨ 1		
LJT	13.3.26	D0	2	offset		1 ⇨ 0		x
LJF	13.3.27	D1	2	offset		1 ⇨ 0		x
LJMP	13.3.28	D2	2	offset				x
CALL	13.3.29	D4	2	FID		n ⇨ 0|1		x
XCALL	13.3.30	D6	2	FID		n ⇨ 0|1		x
PUSH	13.3.31	E0	2	DID		0 ⇨ 1		
PUSHR	13.3.32	E1	2	DID		0 ⇨ 1		
PUSHI	13.3.33	E3	2	value		0 ⇨ 1		
POP	13.3.34	E4	2	DID		1 ⇨ 0	x	
POPR	13.3.35	E5	2	DID		1 ⇨ 0	x	

Tableau 3/T.173 – Synopsis et effets des instructions de la SIR MHEG *(fin)*

Mnémonique	Ref.	Code opérateur (hexa)	Taille du code op	Type du code op	Types des paramètres	Effet sur PS	Effet sur VT	Effet sur le contrôle de flux
POPC	13.3.36	E6	2	DID		1 ⇨ 0	x	
ALLOC	13.3.37	E8	2	TID		0 ⇨ 1	x	
INC	13.3.38	EA	2	DID		1 ⇨ 0	x	
DEC	13.3.39	EB	2	DID		1 ⇨ 0	x	
GET	13.3.40	F0	3	DID, idx		idx ⇨ 1		
GETC	13.3.41	F2	3	DID, idx		idx+1 ⇨ 0	x	
SET	13.3.42	F4	3	DID, idx		idx+1 ⇨ 0	x	
SETC	13.3.43	F6	3	DID, idx		idx+1 ⇨ 0	x	

13.3 Description des instructions

13.3.1 Pas d'opération (no operation)

Description succincte: ne fait rien.

Synopsis: **NOP**

Opérandes: aucun.

Types: sans objet.

Paramètres: aucun.

Pile: ... ⇨ ...

Effet: aucun.

Spécification formelle: **0;**

Erreurs:

13.3.2 Remise (Yield)

Description succincte: traite les messages en attente.

Synopsis: **YIELD**

Opérandes: aucun.

Pile: ... ⇨ ...

Types: sans objet.

Paramètres: aucun.

Effet: s'il y a un message en attente dans la file d'attente des messages, il le traite en appelant la routine correspondante. Ensuite, réitère le traitement jusqu'à obtention d'une file d'attente des messages vide.

Spécification formelle:

```
while (QP != 'null')
{
        FID fid = HT[MQ[QP].MID].FID;
        if (fid == 'null') then    raise('HandlerNotFound');
        else
        {
            CS.push({IP-1, FR, SP, LT});
            // IP-1: allows to re-iterate the YIELD          instruction
            FR = fid;
            IP = RT[FR].IP;
            LT = MQ[QP].LT;
        }
        MQ.remove();
}
```

Erreurs: **HandlerNotFound**

13.3.3 Retour (Return)

Description succincte: retour à l'appelant

Synopsis: **RET**

Opérandes: aucun.

Pile: **..., (Val) ⇨ ..., (Val)**

Types: sans objet.

Paramètres: si la signature de la routine courante possède une valeur de retour, **Val** sera interprétée comme le type de cette valeur de retour.

Dans le cas contraire, on ne s'intéressera à aucun paramètre de pile.

Effet: retour à la routine appelante. Empile la pile d'appel et restaure le contexte de la trame précédente. Si la routine courante a une valeur de retour, vérifie qu'une valeur du même type est présente sur le sommet de la pile des paramètres. Si aucun retour n'est prévu vers une fonction appelante, arrête l'exécution et revient à l'état **ready**.

Spécification formelle:

```
if (sizeof(RT[FR].TID) != (SP - CS[FP].SP))
        then raise('InvalidReturnValue');
IP = CS[FP].IP;
FR = CS[FP].FR;
LT = CS[FP].LT;
CS.pop();
```

Erreurs: **InvalidReturnValue**

13.3.4 Libération (Free)

Description succincte: libère une variable dynamique.

Synopsis: **FREE**

Opérandes: sans objet.

Pile: **..., Did ⇨ ...**

Types: Sans objet.

Paramètres: **Did** sera interprété comme un identificateur de donnée.

Effet:	vérifie que **Did** est l'identificateur de donnée de la variable dynamique. Libère la variable dynamique associée à **Did** et rend non valide l'identificateur de donnée.
Spécification formelle:	**if (did < 8100h) then raise('InvalidParameter');** **delete(VT[PS.pop('data identifier')]);**
Erreurs:	**StackUnderflow** **InvalidIdentifier**

13.3.5 Non (Not)

Description succincte:	négation logique.
Synopsis:	**NOT_<T>**
Opérandes:	aucun.
Pile:	**..., Val ⇨ ..., Neg**
Types:	**Booléen** ou tout type d'entier non signé (B, O, W, U).
Paramètres:	**Val** sera interprété en tant que type <**T**>. **Neg** sera de type <**T**>.
Effet:	remplace l'élément du sommet de la pile des paramètres par sa négation logique. Si <**T**> est égal à **B**, par son complément à 1. **Neg = ~Val**
Spécification formelle:	**type(<T>) buf = PS.pop(<T>);** **if (<T> == 'boolean') then PS.push(! buf);** **else PS.push(~ buf);**
Erreurs:	**StackUnderflow**

13.3.6 Ou (Or)

Description succincte:	disjonction logique.
Synopsis:	**OR_<T>**
Opérandes:	aucun.
Pile:	**..., Val1, Val2 ⇨ ..., Disj**
Types:	**Booléen** ou tout entier non signé (B, O, W, U).
Paramètres:	**Val1** et **Val2** seront interprétés comme étant de type <**T**>. **Disj** sera de type <**T**>.
Effet:	remplace les deux éléments du sommet de la pile des paramètres par leur disjonction logique; Si <**T**> est égal à **B**, par leur disjonction bit-à-bit: **Disj = Val1 | Val2**
Spécification formelle:	**type(<T>) buf = PS.pop(<T>);** **if (<T> == 'boolean') then buf = buf || PS.pop('boolean');** **else buf |= PS.pop(<T>);** **PS.push(buf);**
Erreurs:	**StackUnderflow**

13.3.7 Ou exclusif (Exclusive or)

Description succincte:	exclusion logique.
Synopsis:	**XOR_<T>**
Opérandes:	aucun.
Pile:	**..., Val1, Val2 ⇨ ..., Excl**
Types:	**Booléen** ou tout entier non signé (B, O, W, U).
Paramètres:	**Val1** et **Val2** seront interprétés comme étant de type <**T**>. **Excl** sera de type <**T**>.
Effet:	remplace les deux éléments du sommet de la pile des paramètres par leur exclusion logique; Si <**T**> est égal à **B**, par leur exclusion bit-à-bit: **Excl = Val1 ^ Val2**
Spécification formelle:	**type(<T>) buf = PS.pop(<T>);** **if (<T> == 'boolean') then buf = (buf != PS.pop('boolean'));** **else buf ^= PS.pop(<T>);** **PS.push(buf);**
Erreurs:	**StackUnderflow**

13.3.8 Et (And)

Description succincte:	conjonction logique.
Synopsis:	**AND_<T>**
Opérandes:	aucun.
Pile:	**..., Val1, Val2 ⇨ ..., Conj**
Types:	**Booléen** ou tout entier non signé (B, O, W, U).
Paramètres:	**Val1** et **Val2** seront interprétés comme étant de type <**T**>. **Conj** sera de type <**T**>.
Effet:	remplace les deux éléments du sommet de la pile des paramètres par leur conjonction logique; si <**T**> est de type **B**, par leur conjonction bit-à-bit: **Conj = Val1 & Val2**
Spécification formelle:	**type(<T>) buf = PS.pop(<T>);** **if (<T> == 'boolean') then buf = buf && PS.pop('boolean');** **else buf &= PS.pop(<T>);** **PS.push(buf);**
Erreurs:	**StackUnderflow**

13.3.9 Egalité de référence (Equal reference)

Description succincte:	compare des valeurs construites.
Synopsis:	**EQR**
Opérandes:	aucun
Pile:	**..., Did1, Did2 ⇨ ..., Bool**

Types:	sans objet.
Paramètres:	**Did1** et **Did2** seront interprétés comme des types **data identifier**. **Bool** sera de type **booléen**.
Effet:	vérifie que **Did1** et **Did2** identifient des données de même type. Renvoie la valeur "true" si les données identifiées par **Did1** et **Did2** sont égales (voir 8.2), sinon renvoie la valeur 'false': **Bool = (DT(Did1) == DT(Did2))**
Spécification formelle:	**DID did2 = PS.pop('data identifier');** **DID did1 = PS.pop('data identifier');** **if (DT(did1).tid != DT(did2).tid) then raise('TypeMismatch');** **if (DT(did1).val == DT(did2).val) then PS.push('true');** **else PS.push('false')**
Erreurs:	**TypeMismatch** **StackUnderflow** **InvalidIdentifier**

13.3.10 Egal (Equal)

Description succincte:	égalité.
Synopsis:	**EQ_<T>**
Opérandes:	aucun.
Pile:	**..., Val1, Val2 ⇨ ..., Comp**
Types:	tout type de primitive sauf **void** (O, S, L, W, U, F, D, B, C, I, R)
Paramètres:	**Val1** et **Val2** seront interprétés comme étant detype <**T**>. **Comp** sera de type **booléen**.
Effet:	remplace les deux éléments du sommet de la pile des paramètres par 'true' s'ils sont égaux et par 'false' s'ils ne le sont pas: **Comp = (Val1 == Val2)**
Spécification formelle:	**type(<T>) buf = PS.pop(<T>);** **if (buf == PS.pop<T>) then PS.push('true');** **else PS.push('false');**
Erreurs:	**StackUnderflow**

13.3.11 Inférieur à (Less than)

Description succincte:	infériorité stricte.
Synopsis:	**LT_<T>**
Opérandes:	aucun.
Pile:	**..., Val1, Val2 ⇨ ..., Comp**
Types:	**Character** ou tout type numérique (C, O, S, L, W, U, F, D).
Paramètres:	**Val1** et **Val2** seront interprétés comme étant de type <**T**>. **Comp** sera de type **booléen**.

Effet: remplace les deux éléments du sommet de la pile des paramètres par "true" si l'élément du haut est plus grand que le suivant et par 'false' dans le cas contraire:

Comp = (Val1 < Val2)

On utilisera l'ordre numérique pour comparer des caractères.

Spécification formelle:
```
type(<T>) buf = PS.pop(<T>);
if (PS.pop<T> < buf) then PS.push('true');
else PS.push('false');
```

Erreurs: **StackUnderflow**

13.3.12 Supérieur à (Greater than)

Description succincte: supériorité stricte.

Synopsis: **GT_<T>**

Opérandes: aucun.

Pile: **..., Val1, Val2 ⇨ ..., Comp**

Types: **Character** ou tout type numérique (C, O, S, L, W, U, F, D).

Paramètres: **Val1** et **Val2** seront interprétés comme étant de type <**T**>.
Comp sera de type **booléen.**

Effet: remplace les deux éléments du sommet de la pile des paramètres par "true" si l'élément du haut est plus petit que le suivant et par "false" dans le cas contraire:

Comp = (Val1 > Val2)

On utilisera l'ordre numérique pour comparer des caractères.

Spécification formelle:
```
type(<T>) buf = PS.pop(<T>);
if (PS.pop<T> > buf) then PS.push('true');
else PS.push('false');
```

Erreurs: **StackUnderflow**

13.3.13 Addition (Add)

Description succincte: addition arithmétique.

Synopsis: **ADD_<T>**

Opérandes: aucun.

Pile: **..., Num1, Num2 ⇨ ..., Sum**

Types: tout type numérique (O, S, L, W, U, F, D).

Paramètres: **Num1** et **Num2** seront interprétés comme étant de type <**T**>.
Sum sera de type <**T**>.

Effet: remplace les deux éléments du sommet de la pile des paramètres par leur somme:

Sum = Num1 + Num2

Spécification formelle:
```
type(<T>) buf = PS.pop(<T>);
buf += PS.pop(<T>);
PS.push(buf);
```

Erreurs: **StackUnderflow**
ArithmeticOverflow

13.3.14 Soustraction (Subtract)

Description succincte: soustraction arithmétique.

Synopsis: **SUB_<T>**

Opérandes: aucun.

Pile: **..., Num1, Num2 ⇨ ..., Diff**

Types: tout type numérique (O, S, L, W, U, F, D).

Paramètres: **Num1** et **Num2** seront interprétés comme étant de type <**T**>.
Diff sera de type <**T**>.

Effet: remplace les deux éléments du sommet de la pile des paramètres par leur différence:
Diff = Num1 - Num2

Spécification formelle:
```
type(<T>) buf = PS.pop(<T>);
buf = PS.pop(<T>) –buf;
PS.push(buf);
```

Erreurs: **StackUnderflow**
ArithmeticOverflow

13.3.15 Multiplication (Multiply)

Description succincte: multiplication arithmétique.

Synopsis: **MUL_<T>**

Opérandes: aucun.

Pile: **..., Num1, Num2 ⇨ ..., Prod**

Types: tout type numérique (O, S, L, W, U, F, D).

Paramètres: **Num1** et **Num2** seront interprétés comme étant de type <**T**>.
Prod sera de type <**T**>.

Effet: remplace les deux éléments du sommet de la pile des paramètres par leur produit
Prod = Num1 * Num2

Spécification formelle:
```
type(<T>) buf = PS.pop(<T>);
buf *= PS.pop(<T>);
PS.push(buf);
```

Erreurs: **StackUnderflow**
ArithmeticOverflow

13.3.16 Division (Divide)

Description succincte: division arithmétique.

Synopsis: **DIV_<T>**

Opérandes: aucun.

Pile:	**..., Num1, Num2 ⇨ ..., Quot**
Types:	tout type numérique (O, S, L, W, U, F, D).
Paramètres:	**Num1** et **Num2** seront interprétés comme étant de type <**T**>. **Quot** sera de type <**T**>.
Effet:	remplace les deux éléments du sommet de la pile des paramètres par leur quotient: **Quot = Num1/Num2**
Spécification formelle:	**type(<T>) buf = PS.pop(<T>);** **buf = PS.pop(<T>) / buf;** **PS.push(buf);**
Erreurs:	**StackUnderflow** **DivisionByZero**

13.3.17 Négation (Negate)

Description succincte:	changement de signe.
Synopsis:	**NEG_<T>**
Opérandes:	aucun.
Pile:	**..., Num ⇨ ..., Opp**
Types:	tout numérique signé (S, L, F, D).
Paramètres:	**Num** sera interprété comme étant de type <**T**>. **Opp** sera de type <**T**>.
Effet:	remplace l'élément du sommet de la pile des paramètres par son opposé: **Opp = –Num1**
Spécification formelle:	**type(<T>) buf = PS.pop<T>;** **PS.push(–buf);**
Erreurs:	**StackUnderflow**

13.3.18 Reste (Remainder)

Description succincte:	reste arithmétique.
Synopsis:	**REM_<T>**
Opérandes:	aucun.
Pile:	**..., Num1, Num2 ⇨ ..., Rem**
Types:	tout entier (O, S, L, W, U).
Paramètres:	**Num1** et **Num2** seront interprétés comme étant de type <T>. **Rem** sera de type <**T**>.
Effet:	remplace les deux éléments du sommet de la pile des paramètres par leur reste: **Rem = Num1 % Num2**
Spécification formelle:	**type(<T>) buf = PS.pop(<T>);** **buf = PS.pop(<T>) % buf;** **PS.push(buf);**

Erreurs:	**StackUnderflow** **DivisionByZero**

13.3.19 Duplication (Duplicate)

Description succincte:	duplication de valeur.
Synopsis:	**DUP_<T>**
Opérandes:	sans objet.
Pile:	**..., Val ⇨ ..., Val, Val**
Types:	tout type de primitive sauf **void** (O, S, L, W, U, F, D, B, C, I, R)
Paramètres:	**Val** sera interprétée comme étant de type <**T**>.
Effet:	duplique la valeur sur le sommet de la pile.
Spécification formelle:	**type(<T>) buf = PS[SP](<T>);** **PS.push(buf);**
Erreurs:	**StackUnderflow**

13.3.20 Conversion (Convert)

Description succincte:	conversion de valeur.
Synopsis:	**CVT_<T1><T2>**
Opérandes:	sans objet.
Pile:	**..., Val ⇨ ..., Res**
Types:	**Booléen, caractère** ou tout type numérique (O, S, L, W, U, F, D, B, C), voir les combinaisons autorisées au 13.4.
Paramètres:	**Val** sera interprétée comme étant de type <**T1**> (type source). **Res** sera de type <**T2**> (type destination).
Effet:	remplace la valeur du sommet de pile par une valeur équivalente dans le type destination. Les conversions de règles définies au 13.4 sont applicables.
Spécification formelle:	**type(<T2>) buf = (type(<T2>)) (PS.pop(<T1>));** **PS.push(buf);**
Erreurs:	**StackUnderflow**

13.3.21 Saut sur condition vraie (Jump on true)

Description succincte:	"si" de saut conditionnel court.
Synopsis:	**JT Off**
Opérandes:	**Off** sera un offset signé tenant sur un octet (en notation de complément à deux) spécifiant le nombre de décalages d'instructions en avant ou en arrière à l'intérieur de la routine courante.
Pile:	**..., Test ⇨ ...**
Types:	sans objet.

Paramètres: **Test** sera interprété comme étant de type Booléen..

Effet: si l'élément du sommet de pile est à 'true' alors
si **Off** est positif, alors saut en avant de **Off** instructions;
si **Off** est négatif, alors saut en arrière de **Off** instructions; .

Spécification formelle: **if (PS.pop('boolean')) then IP += Off;**

Erreurs: **StackUnderflow**
JumpOutOfRange

13.3.22 Saut sur condition fausse (Jump on false)

Description succincte: "sinon" de saut conditionnel court.

Synopsis: **JF Off**

Opérandes: **Off** sera un offset signé tenant sur un octet (en notation de complément à deux) spécifiant le nombre de décalages d'instructions en avant ou en arrière à l'intérieur de la routine courante.

Pile: **..., Test ⇨ ...**

Types: sans objet.

Paramètres: **Test** sera interprété comme étant de type **Booléen.**.

Effet: si l'élément du sommet de pile est à 'false' alors
si **Off** est positif, alors saut en avant de **Off** instructions;
si **Off** est négatif, alors saut en arrière de **Off** instructions; .

Spécification formelle: **if ! (PS.pop('boolean')) then IP += Off;**

Erreurs: **StackUnderflow**
JumpOutOfRange

13.3.23 Saut (Jump)

Description succincte: saut inconditionnel.

Synopsis: **JMP Off**

Opérandes: **Off** sera un offset signé tenant sur un octet (en notation de complément à deux) spécifiant le nombre de décalages d'instructions en avant ou en arrière à l'intérieur de la routine courante.

Pile: ... ⇨ ...

Types: sans objet.

Paramètres: aucun.

Effet: si **Off** est positif, saut de **Off** instructions en avant;
si **Off** est négatif, saut de **Off** instructions en arrière.

Spécification formelle: **IP += Off;**

Erreurs: **JumpOutOfRange**

13.3.24 Décalage (Shift)

Description succincte:	décalage.
Synopsis:	**SHIFT_<T> Off**
Opérandes:	**Off** sera un offset signé tenant sur un caractère (en notation de complément à deux) spécifiant le nombre de décalages de bits du paramètre à effectuer vers la gauche ou la droite.
Pile:	**..., Val ⇨ ..., Pwr**
Types:	tout entier non signé (O, W, U).
Paramètres:	**Val** sera interprété comme étant de type <**T**>. **Pwr** sera de type <**T**>.
Effet:	remplace l'élément du sommet de la pile par sa valeur décalée de **Off** bits vers la droite si **Off** est positif, ou vers la gauche de -**Off** bits si **Off** est négatif. Si **Off** est en dehors de l'intervalle, le résultat sera indéterminé..
Spécification formelle:	**type(<T>) buf = PS.pop(<T>);** **if (Off >=0) then buf >>= Off;** **else if (buf < 0)** **buf = –(–buf << –Off);** **else buf <<= –Off;** **PS.push(buf);**
Erreurs:	**StackUnderflow** **ShiftOutOfRange**

13.3.25 Extraction de références à objet (Get object reference)

Description succincte:	extrait du bloc la référence à objet initiale.
Synopsis:	**GETOR Pid**
Opérandes:	**Pid** sera une représentation sur un caractère d'un identificateur de bloc spécifiant le bloc à accéder.
Pile:	**... ⇨ ..., Obref**
Types:	sans objet.
Paramètres:	**Obref** sera de type **object reference** .
Effet:	récupère une référence à objet pour l'objet initial du bloc.
Spécification formelle:	**if (PT[PID].sts = 'not available') then raise('BadPackageStatus');** **PS.push(PT[PID].or);**
Erreurs:	**InvalidIdentifier** **BadPackageStatus**

13.3.26 Saut long sur condition vraie (Long jump on true)

Description succincte:	"si" de saut conditionnel long.
Synopsis:	**LJT Off**

Opérandes: **Off** sera un offset signé tenant sur deux octets (en notation de complément à deux) spécifiant le nombre de décalages d'instructions en avant ou en arrière à l'intérieur de la routine courante.

Pile: **..., Test ⇨ ...**

Types: sans objet.

Paramètres: **Test** sera interprété comme étant de type **Booléen**.

Effet: si l'élément du sommet de pile est à 'true' alors
si **Off** est positif, alors saut en avant de **Off** instructions;
si **Off** est négatif, alors saut en arrière de **Off** instructions; .

Spécification formelle:

```
if (PS.pop('boolean')) then IP += Off;
```

Erreurs: **StackUnderflow**
JumpOutOfRange

13.3.27 Saut long sur condition fausse (Long jump on false)

Description succincte: "else" de saut conditionnel long.

Synopsis: **LJF Off**

Opérandes: **Off** sera un offset signé tenant sur un octet (en notation de complément à deux) spécifiant le nombre de décalages d'instructions en avant ou en arrière à l'intérieur de la routine courante.

Pile: **..., Test ⇨ ...**

Types: sans objet.

Paramètres: **Test** sera interprété comme étant de type **Booléen**.

Effet: si l'élément du sommet de pile est à 'false' alors
si **Off** est positif, alors saut en avant de **Off** instructions;
si **Off** est négatif, alors saut en arrière de **Off** instructions; .

Spécification formelle:

```
if ! (PS.pop('boolean')) then IP += Off;
```

Erreurs: **StackUnderflow**
JumpOutOfRange

13.3.28 Saut long (Long jump)

Description succincte: saut inconditionnel long.

Synopsis: **LJMP Off**

Opérandes: **Off** sera un offset signé tenant sur un octet (en notation de complément à deux) spécifiant le nombre de décalages d'instructions en avant ou en arrière à l'intérieur de la routine courante.

Pile: **... ⇨ ...**

Types: sans objet.

Paramètres: aucun.

Effet: Si **Off** est positif, saut de **Off** instructions en avant;
Si **Off** est négatif, saut de **Off** instructions en arrière.

Spécification formelle:
```
IP += Off;
```

Erreurs: **JumpOutOfRange**

13.3.29 Appel (Call)

Description succincte: appel de routine.

Synopsis: **CALL Fid**

Opérandes: **Fid** sera la représentation sur deux caractères d'un identificateur de fonction spécifiant la routine à invoquer.

Pile: **..., ParN, ... , Par1 ⇨ ...**

Types: sans objet.

Paramètres: **Par1, ..., ParN** (où N est le nombre de paramètres de la routine) sont les paramètres effectifs de la routine. Ils seront interprétés comme étant du même type que les paramètres formels de la routine lorsque ceux-ci sont passés par valeur. Ils seront interprétés comme étant de types **data identifier** et référenceront une variable du même type que celui des paramètres formels de la routine lorsque ceux-ci sont passés par référence.

Effet: empile les éléments sur le sommet de la pile des paramètres et invoque la routine spécifiée dans Fid avec ces éléments comme paramètres formels. Pour les paramètres passés par référence, vérifie que l'identificateur de données ne référence pas une variable locale et pointe sur des données d'un type identique à celui de la signature. Empile une trame sur la pile d'appel avec le contexte courant. Initialise la table des variables locales de la routine. Met le pointeur d'instruction sur le première instruction de la routine.

Spécification formelle:
```
TID tid;
CS.push(IP, FR, SP, LT);
FR = Fid;
LT = RT[Fid].LT;
for (short i = 0; i < RT[Fid].nbp; i--;)
{
        switch (RT[Fid].sig[i].mod)
        {
        case 'value':
            tid = RT[Fid].sig[i].TID;
            break;
        case 'reference':
            tid == 'data identifier';
```

```
                    if (8000h <= PS[SP](tid) < 8100h)
                    then raise('InvalidParameter');
                    if (RT[Fid].sig[i].TID !=
                    DT(PS[SP](tid)).TID)
                    then raise('TypeMismatch');
                    break;
                };
                LT[I+0x8000].val = PS.pop(tid);
};
IP = RT[Fid].IP;
```

Erreurs: **InvalidIdentifier**
StackUnderflow
TypeMismatch
InvalidParameter

13.3.30 Appel externe (External call)

Description succincte: appel de fonction externe.

Synopsis: **XCALL Fid**

Opérandes: **Fid** sera la représentation sur deux caractères d'un identificateur de fonction spécifiant le service ou la fonction prédéfinie à invoquer.

Pile: **..., ParN, ... , Par1 ⇨ ..., (Ret)**

Types: sans objet.

Paramètres: **Par1** sera interprété comme étant de type **object reference**. Il indique la référence à l'instance IDL à laquelle s'applique l'opération. **Par2, ..., ParN** (où N est le nombre de paramètres de la fonction + 1) sont les paramètres effectifs de la fonction. Ils seront interprétés comme étant de type **data identifier** quelque soit le mode de passage.
Si la fonction possède une valeur de retour autre que **void, Ret** sera du type de la valeur de retour.

Effet: vérifie que les paramètres référencent des données de même type que ceux de la signature de la fonction. Empile les éléments sur le sommet de la pile des paramètres et invoque la fonction externe spécifiée dans Fid avec ces éléments comme paramètres effectifs. Empile une trame sur la pile d'appel avec le contexte courant. Passe les paramètres à la fonction externe et l'invoque. Si l'invocation est asynchrone, dépile la pile d'appel dès que l'accusé de réception de la requête est reçu. Si l'invocation est synchrone, attend que la requête se termine. Si une exception apparaît, active le filet de l'exception. Sinon, récupère les paramètres de sortie de la fonction ainsi que la valeur de retour, empile la valeur de retour sur la pile des paramètres et dépile la pile d'appel.

Spécification formelle:

```
DID buf[ST[Fid].nbp];
Object obref = PS.pop('object reference');
for (short i = 0; i < ST[Fid].nbp; i++;)
{
        if (ST[Fid].sig[i].TID != DT(PS[SP]('data identifier').TID))
        then raise('TypeMismatch');
        buf[I] = PS.pop('data identifier');
};
```

```
CS.push(IP, FR, SP, LT);
FR = Fid;
LT[0].tid = 'object reference';
LT[0].val = obref;
for (short i = 1;i < ST[Fid].nbp; i++}
        LT[i].TID = 'data identifier';
        LT[i].val = buf[i];
}
short Pid = (Fid>>8)–64;
if (PT[Pid].sts != 'available') then raise('BadPackageStatus');
_open_package(PT[Pid].name);

// open a context to invoke the service
_pass_in_parameter(LT[0]);
// according to the platform mapping specification procedure

for (short i=1; i<ST[Fid].nbp; i++;)
        switch(ST[Fid].sig[i].mod)
        {
        case 'in': _pass_in_parameter (LT[i]);
        case 'out': _pass_out_parameter (LT[i]);
        case 'inout': _pass_inout_parameter (LT[i]);
        };

if (ST[Fid].mod == 'asynchronous') then
{
        _invoke_operation(PT[Pid].name, ST[Fid].name);
        {IP, FR, SP, LT} = CS.pop();
}

else
{
        result = _invoke_operation(PT[Pid].name, ST[Fid].name);
        if (result == 'ok')          then
        {
            _retrieve_out_parameter();
            {IP, FR, SP, LT} = CS.pop();
            if (ST[Fid].TID != 'void') then
                PS.push(_retrieve_return_value());
        }

        else  // the result is an exception formatted as a message
        {
            FR = HT[result.MID];
            LT = result.LT;
            IP = RT[FT].IP;
        }
}

_close_package(PT[Pid].name);
// close service invocation context
```

Erreurs: **InvalidIdentifier**
StackUnderflow
TypeMismatch
BadPackageStatus
InvalidObjectReference

13.3.31 Empiler (Push)

Description succincte: empile une valeur de donnée.

Synopsis: **PUSH Did**

Opérandes: **Did** sera la représentation tenant sur deux octets d'un identificateur de donnée contenant la valeur à empiler.

Pile: **... ⇨ ..., Val**

Types: sans objet.

Paramètres: **Val** sera d'un type identique à celui de la constante ou la variable identifiée par **Did**.

Effet: vérifie que **Did** identifie une constante ou une variable d'un type primitif. Empile sur la pile des paramètres la valeur de la constante ou de la variable dont l'identificateur de donnée est **Did**.

Spécification formelle:
```
if (DT(Did).tid > 'object reference') then raise('InvalidType');
PS.push(DT(Did).val);
```

Erreurs: **InvalidIdentifier**
InvalidType

13.3.32 Empiler référence (Push reference)

Description succincte: empile un identificateur de donnée.

Synopsis: **PUSHR Did**

Opérandes: **Did** sera la représentation tenant sur deux octets d'un identificateur de donnée contenant la valeur à empiler.

Pile: **... ⇨ ..., Val**

Types: sans objet.

Paramètres: **Val** sera de type **data identifier**.

Effet: empile **Did** dans la pile de paramètres.

Spécification formelle:
```
PS.push(Did);
```

Erreurs: aucun.

13.3.33 Empiler valeur immédiate (Push immediate)

Description succincte: empile un entier court.

Synopsis: **PUSHI Int**

Opérandes: **Int** sera la représentation sur deux octets d'une valeur entière courte (en notation complément à deux) spécifiant la valeur à empiler.

Pile: **... ⇨ ..., Val**

Types: sans objet.

Paramètres: **Val** sera de type short.

Effet: empile **Int** dans la pile de paramètres.

Spécification formelle:
```
PS.push(Int);
```

Erreurs: aucun.

13.3.34 Dépiler (Pop)

Description succincte:	dépile une valeur et l'affecte à la variable.
Synopsis:	**POP Did**
Opérandes:	**Did** sera la représentation sur deux octets de l'identificateur de donnée de la variable à laquelle l'élément du sommet de pile sera affecté.
Pile:	**..., Val ⇨ ...**
Types:	sans objet.
Paramètres:	**Val** sera interprété comme le type de la variable identifiée par **Did**.
Effet:	vérifie que **Did** identifie une variable de type primitif. Dépile **Val** de la pile de paramètres et l'affecte à la variable identifiée par **Did**.
Spécification formelle:	**TID tid = VT(Did).TID;** **if (tid > 'object reference') then raise('InvalidType')** **VT(Did).val = PS.pop(tid);**
Erreurs:	**InvalidIdentifier** **StackUnderflow** **InvalidType**

13.3.35 Dépiler référence (Pop reference)

Description succincte:	dépile une valeur et l'affecte à la variable référencée par l'opérande.
Synopsis:	**POPR Did**
Opérandes:	**Did** sera la représentation sur deux octets de l'identificateur de donnée de la variable de type **data identifier** dont la valeur identifie la variable à laquelle l'élément du sommet de pile sera affecté.
Pile:	**..., Val ⇨ ...**
Types:	sans objet.
Paramètres:	**Val** sera interprétée comme étant de type **VT(Did).val**.
Effet:	vérifie que **Did** identifie une variable de type **data identifier**. Dépile **Val** de la pile de paramètres et l'affecte à la variable identifiée par **Did**..
Spécification formelle:	**TID tid = type(VT(Did).val.TID);** **if (tid != 'data identifier') then raise('InvalidType');** **VT(VT(Did).val).val = PS.pop(tid);**
Erreurs:	**InvalidIdentifier** **StackUnderflow** **InvalidType**

13.3.36 Dépiler contenus (Pop contents)

Description succincte:	dépile une variable et affecte sa valeur à une variable.
Synopsis:	**POPC Did1**
Opérandes:	**Did1** sera la représentation sur deux octets de l'identificateur de donnée de la variable à laquelle la valeur de la donnée identifiée par l'élément du sommet de pile sera affectée.

Pile: **..., Did2 ⇨ ...**

Types: sans objet.

Paramètres: **Did2** sera interprétée comme étant de type **data identifier**. La donnée identifiée par **Did2** sera interprétée comme étant du type de la donnée identifiée par **Did1**.

Effet: vérifie que **Did1** et **Did2** identifient des données de même type.
Dépile **Did2** de la pile des paramètres et affecte la valeur de **Did2** à la variable identifiée par **Did1**.

Spécification formelle:
```
DID did2 = PS.pop('data identifier');
if (VT(Did1).TID != DT(did2).TID) then raise('TypeMismatch');
VT(Did1).val = DT(did2).val;
```

Erreurs: **InvalidIdentifier**
StackUnderflow
TypeMismatch

13.3.37 Allocation (Allocate)

Description succincte: crée une variable dynamique.

Synopsis: **ALLOC Tid**

Opérandes: **Tid** sera la représentation sur deux octets d'un identificateur de type spécifiant la type de la valeur dynamique à allouer.

Pile: **..., ⇨ ..., Did**

Types: sans objet.

Paramètres: **Did** sera de type **data identifier**.

Effet: génère une variable dynamique dont le type est identifié par **Tid**. Empile son identificateur de donnée dans la pile des paramètres.

Spécification formelle:
```
DID did = new(Tid);
VT[did].val = 'null';     // default value for the type
VT[did].TID = Tid;
PS.push(did);
```

Erreurs: **AllocationFailed**
InvalidIdentifier

13.3.38 Incrément (Increment)

Description succincte: incrémente une variable.

Synopsis: **INC Did**

Opérandes: **Did** sera la représentation sur deux octets de l'identificateur de donnée de la variable à incrémenter.

Pile: **..., Val ⇨ ...,**

Types: sans objet.

Paramètres: **Val** sera interprété comme étant du type de la variable identifiée par **Did**.

Effet: vérifie que **Did** identifie une variable de type numérique. Empile et incrémente la valeur de la variable identifiée par **Did** avec la valeur dépilée.

Spécification formelle:
```
TID tid = VT(Did).TID;
if (VT(Did).TID > 'double') raise('InvalidType');
VT(Did).val += PS.pop(<T>);
```

Erreurs: **InvalidIdentifier**
StackUnderflow
InvalidType
ArithmeticOverflow

13.3.39 Décrément (Decrement)

Description succincte: décrémente une variable.

Synopsis: **DEC Did**

Opérandes: **Did** sera la représentation sur deux octets de l'identificateur de données de la variable à décrémenter. La variable identifiée par **Did** sera de type numérique.

Pile: **..., Val ⇨ ...,**

Types: sans objet.

Paramètres: **Val** sera interprété comme étant du type de la variable identifiée par **Did**.

Effet: vérifie que **Did** identifie une variable de type numérique. Dépile et décrémente la valeur de la variable identifiée par **Did** avec la valeur dépilée.

Spécification formelle:
```
TID tid = VT(Did).TID;
if (VT(Did).TID > 'double') raise('InvalidType');
VT(Did).val -= PS.pop(<T>);
```

Erreurs: **InvalidIdentifier**
StackUnderflow
InvalidType
ArithmeticOverflow

13.3.40 Extraction (Get)

Description succincte: extrait la valeur de l'élément de donnée d'un type construit.

Synopsis: **GET Did Lvl**

Opérandes: **Did** sera la représentation sur deux octets de l'identificateur de donnée de la donnée à accéder.
Lvl sera une quantité non signée tenant sur un octet représentant le nombre de niveaux imbriqués à parcourir pour accéder à la valeur recherchée.

Pile: **..., Idx(Lvl), ..., Idx(1) ⇨ ..., Val**

Types: sans objet.

Paramètres: **Idx(1), ... Idx(Lvl)** seront interprétés comme étant de type **unsigned short**.
Val sera du même type que celui de l'élément accédé.

Effet: remplace la liste d'indices de la pile des paramètres par la valeur de l'élément adressé par les indices dépilés à l'intérieur de la donnée de type construit identifiée par **Did**:

Val = DT(Did)[Idx(1),...,Idx(Lvl)]

vérifie que l'élément accédé est de type primitif. Si **Lvl** est égal à 0, le traite comme une instruction **PUSH**.

Spécification formelle:

```
void *buf = &DT(Did);
unsigned short idx;
for (;Lvl>0; Lvl--;)
{
        if (buf->TID <= 'object reference') then   raise('InvalidLevel');
        idx = PS.pop('unsigned short');
        if (buf->lg < idx) then raise        ('InvalidIndex');
        buf = &buf->.val[idx];
}
if (buf->TID > 'object reference') then raise('InvalidType');
PS.push(buf->val);
```

Erreurs: **InvalidIdentifier**
InvalidLevel
StackUnderflow
InvalidIndex
InvalidType

13.3.41 Extraction de contenus (Get contents)

Description succincte: affecte les contenus de données à un élément de donnée de type construit.

Synopsis: **GETC Did1 Lvl**

Opérandes: **Did1** sera la représentation sur deux octets de l'identificateur de donnée de la donnée à accéder.
Lvl sera une quantité non signée tenant sur un octet représentant le nombre de niveaux imbriqués à parcourir pour accéder à la valeur recherchée.

Pile: **..., Did2, Idx(Lvl), ..., Idx(1)** ⇨ **...,**

Types: sans objet.

Paramètres: **Idx(1), ... Idx(Lvl)** seront interprétés comme étant de type **unsigned short**.

Effet: dépile une liste d'indices et l'identificateur de donnée **Did2** de la pile de paramètres. A l'intérieur de la donnée de type construit identifiée par **Did1**, affecte la variable identifiée par **Did2** à l'élément adressé par la liste d'indices dépilée:

VT(Did2).Val = DT(Did1)[Idx(1),...,Idx(Lvl)]

vérifie que l'élément à accéder est du même type que celui de la donnée identifiée par **Did2**.

Spécification formelle:

```
void *buf = &DT(Did1);
unsigned short idx;
for (;Lvl>0; Lvl--;)
{
        if (buf->TID <= 'object reference') then   raise('InvalidLevel');
        idx = PS.pop('unsigned short');
        if (buf->lg < idx) then raise        ('InvalidIndex');
        buf = &buf->.val[idx];
}
DID did2 = PS.pop('data identifier');
if (VT(did2).TID != buf->TID) then raise('TypeMismatch');
VT(did2).val = buf->val;
```

Erreurs:

InvalidIdentifier
InvalidLevel
StackUnderflow
InvalidIndex
TypeMismatch

13.3.42 Affectation (Set)

Description succincte: affecte une valeur à un élément de variable de type construit.

Synopsis: **SET Did Lvl**

Opérandes: **Did** sera la représentation sur deux octets de l'identificateur de donnée de la variable à modifier.
Lvl sera une quantité non signée tenant sur un octet représentant le nombre de niveaux imbriqués à parcourir pour accéder à la valeur à modifier.

Pile: **..., Val, Idx(Lvl), ..., Idx(1)** ⇨ **...,**

Types: sans objet.

Paramètres: **Idx(1), ... Idx(Lvl)** seront interprétés comme étant de type **unsigned short**.
Val sera interprété comme étant du même type que celui de l'élément à modifier.

Effet: dépile une liste d'indices et une valeur de la pile des paramètres. A l'intérieur de la variable structurée identifiée par **Did**, affecte à la valeur dépilée l'élément adressé par la liste des indices dépilée:

VT(Did) [Idx(1),...,Idx(Lvl) = VAL.

vérifie que l'élément à accéder est de type primitif. Si **Lvl** est égal à 0, le traite comme une instruction **POP**.

Spécification formelle:

```
void *buf = &VT(Did);
unsigned short idx;
for (;Lvl>0; Lvl--;)
{
        if (buf->TID <= 'object reference') then   raise('InvalidLevel');
        idx = PS.pop('unsigned short');
        if (buf->lg < idx) then raise        ('InvalidIndex');
        buf = &buf->.val[idx];
}
if (buf->TID > 'object reference') then raise('InvalidType');
buf->val = PS.pop(buf->TID);
```

Erreurs: **Invalid Identifier**
InvalidLevel
StackUnderflow
InvalidIndex
InvalidType

13.3.43 Affectation de contenus (Set contents)

Description succincte: affecte des contenus de données à un élément de variable de type construit.

Synopsis: **SETC Did1 Lvl**

Opérandes: **Did1** sera la représentation sur deux octets de l'identificateur de donnée de la variable à modifier.
Lvl sera une quantité non signée tenant sur un octet représentant le nombre de niveaux imbriqués à parcourir pour accéder à l'élément à modifier.

Pile: **..., Did2, Idx(Lvl), ..., Idx(1) ⇨ ...,**

Types: sans objet.

Paramètres: **Idx(1), ... Idx(Lvl)** seront interprétés comme étant de type **unsigned short**. **Did2** sera interprété comme étant de type **data identifier**.

Effet: dépile une liste d'indices et une valeur de la pile des paramètres. A l'intérieur de la variable structurée identifiée par **Did1**, affecte à la valeur identifiée par la valeur dépilée, l'élément adressé par la liste des indices dépilée:

VT(Did1)[Idx(1),...,Idx(Lvl)] = DT(Did2).

vérifie que **Did2** identifie une donnée du même type que celui de l'élément à modifier. Si **Lvl** est égal à 0, le traite comme une instruction **POPC**.

Spécification formelle:

```
void *buf = VT(Did1);
unsigned short idx;
for (;Lvl>0; Lvl--;)
{
        if (buf->TID <= 'object reference') then   raise('InvalidLevel');
        idx = PS.pop('unsigned short');
        if (buf->lg < idx) then raise        ('InvalidIndex');
        buf = &buf->.val[idx];
}
DID did2 = PS.pop('data identifier');
if (DT(did2).TID != buf->TID) then raise('TypeMismatch');
buf->val = DT(did2).val;
```

Erreurs: **InvalidIdentifier**
InvalidLevel
StackUnderflow
InvalidIndex
TypeMismatch

13.4 Règles de conversion de types

Le présent sous-paragraphe définit les règles s'appliquant lorsque l'instruction **convert (CVT)** est utilisée pour convertir une valeur de la pile des paramètres (une valeur de type primitif) d'un type source vers un type destination.

Les valeurs des types **data identifier** et **object reference** ne seront pas converties d'une valeur vers un autre type et inversement.

Pour ce qui concerne les autres types primitifs, toutes les conversions ne sont pas autorisées; cependant, n'importe quel type peut être converti dans un autre en appliquant des conversions en cascades.

Le Tableau 4 montre les conversions de types autorisées ainsi que les numéros de sous-paragraphes où elles sont définies.

Tableau 4/T.173 – Conversions de types

Source/Destination	O	S	L	W	U	F	D	B	C
O	N/A	13.4.2.2	N/A	13.4.2.2	N/A	N/A	N/A	13.4.4	N/A
S	N/A	N/A	13.4.2.3	13.4.1	13.4.3	N/A	N/A	13.4.4	N/A
L	N/A	13.4.5	N/A	N/A	13.4.1	13.4.2.3	N/A	13.4.4	N/A
W	13.4.5	13.4.1	13.4.2.3	N/A	13.4.2.3	N/A	N/A	13.4.4	13.4.1
U	N/A	N/A	13.4.1	13.4.5	N/A	13.4.2.3	N/A	13.4.4	N/A
F	N/A	N/A	13.4.6	N/A	13.4.6	N/A	13.4.2.3	N/A	N/A
D	N/A	N/A	N/A	N/A	N/A	13.4.5	N/A	N/A	N/A
B	13.4.2.1	13.4.2.1	N/A	N/A	N/A	N/A	N/A	N/A	N/A
C	N/A	N/A	N/A	13.4.1	N/A	N/A	N/A	N/A	N/A
N/A sans objet.									

13.4.1 Conversions réversibles

Les conversions suivantes sont sans perte (préservent l'information) en cas de réversibilité:

– entre **unsigned short** et **character** (WC, CW);

– entre **short** et **unsigned short** (SW, WS);

– entre **long** et **unsigned long** (LU, UL).

Pour toutes ces conversions, le résultat de la conversion sera la valeur du type destination ayant la même notation en complément à deux que celle de la valeur source.

13.4.2 Extensions sans perte

Les conversions suivantes étendent la valeur source sans provoquer de perte:

– de **boolean** (BO, BS) (voir 13.4.2.1);

– de **octet** vers un type numérique (OS, OW) (voir 13.4.2.2);

– d'un type numérique signé vers un autre type numérique signé avec un intervalle plus grand (SL, LF, FD) (voir 13.4.2.3);

– d'un type numérique non signé vers tout autre type numérique avec un intervalle plus grand (WL, WU, UF) (voir 13.4.2.3).

13.4.2.1 Conversion de booléen

Si la valeur du **boolean** source est **false**, sa valeur dans le type destination sera égale à 0.

Si la valeur du **boolean** source est **true**, sa valeur dans le type destination sera la valeur correspondant à tous les bits mis à 1 (en notation complément à deux), à savoir:

– 255 pour une destination de type **octet**;

– –1 pour une destination de type **short**.

13.4.2.2 Conversion d'octet vers un type numérique

La valeur dans le type destination sera celle de l'octet.

13.4.2.3 Conversion sans perte d'un type numérique vers un type numérique plus grand

La valeur dans le type destination sera la même valeur numérique que celle du type source.

13.4.3 Extensions avec perte

La conversion de **short** à **unsigned long** (SU) se traitera comme suit:

– si la valeur source est positive ou nulle, la valeur destination sera la même valeur numérique que la valeur source;

– si la valeur source est strictement négative, la valeur destination est indéterminée.

13.4.4 Troncations vers booléens

Les troncations de types **octet** ou numérique vers un type **boolean** (OB, SB, WB, LB, UB) se traiteront de la manière suivante:

– si la valeur de la source est 0, la valeur de la destination sera **false**;

– si la valeur de la source est différente de 0, la valeur de la destination sera **true**.

13.4.5 Troncations entre types entiers ou flottants

Les troncations d'un type entier vers un type **octet** ou entier (WO, LS, UW) ou, d'un type en virgule flottante vers un autre type en virgule flottante (DF) se traiteront de la manière suivante:

– si la valeur source est comprise dans l'intervalle du type destination, alors la valeur destination sera la même valeur numérique que la valeur source,

– sinon la valeur destination est indéterminée.

13.4.6 Troncations de valeur flottante à entier

Les troncations d'un type floating-point vers un type integer (FL, FU) se traiteront de la manière suivante:

– premièrement, la partie décimale de la valeur source sera tronquée et changée en valeur entière (arrondie à la valeur immédiatement inférieure);

– ensuite, les règles définies au 13.4.5 s'appliqueront à la valeur tronquée.

14 Mappage entre la SIR MHEG et IDL

Le présent paragraphe spécifie comment une opération IDL pourra être mappée avec les déclarations d'un script échangé, lorsque cette spécification IDL aura pour objectif d'être utilisée par le script comme fournissant un service externe.

Le présent paragraphe définit le mappage vers les déclarations de la SIR MHEG pour les:

– interfaces et modules IDL;

– types IDL;

– constantes IDL;

– références aux objets IDL;

– opérations IDL;

– attributs IDL;

– exceptions IDL.

14.1 Spécifications IDL

A une spécification IDL correspondra un **PackageDeclaration** de la SIR MHEG déclaré comme un composant d'un composant **external-package-declarations** de **InterchangedScript**. Le nom de la spécification IDL correspondra au composant **name** de cette déclaration de bloc.

NOTE – Des exemples de spécification IDL sont l'API MHEG, MPEG/DSM-CC.

Si le nombre d'opérations ou d'exceptions d'une spécification IDL va au delà de la taille d'un bloc, la spécification sera décomposée en plusieurs blocs de même nom ayant différents identificateurs SIR MHEG.

14.2 Modules et interfaces IDL

Comme la déclaration de bloc possède une organisation de type rateau, il n'y a de mappage ni pour un module, ni pour une interface IDL. Cependant, une référence à l'interface encapsulant (à savoir un paramètre de type **Object**) sera fournie sous la forme d'un paramètre implicite à chaque invocation de fonction décrivant une opération IDL.

14.3 Opérations IDL

A une opération IDL mappera un composant **services** de la déclaration de bloc de la SIR MHEG, faisant mapper la spécification IDL à laquelle appartient l'opération.

14.3.1 Nom d'opération

Au nom global d'une opération IDL correspondra le composant **name** de la SIR MHEG pour cette description de service.

14.3.2 Paramètres d'opération

Aux paramètres d'une opération IDL correspondra le composant **parameters-description** de la description de service. Dans un **ServiceParameterDescription** correspondra à chaque type de paramètre IDL un composant **type** identifiant un type déclaré conformément aux règles de mappage définies dans le présent paragraphe. Le mode IDL de passage de paramètre correspondra au composant **passing-mode** de la description du paramètre de service correspondant de la SIR MHEG.

Si l'opération ne possède ni paramètre de sortie, ni valeur de retour et est plus spécialement décrite pour renvoyer plusieurs exceptions de manière consécutive (à des fins de notification), la valeur de son composant **calling-mode** devrait être "**asynchronous**". Sinon, sa valeur sera "**synchronous**".

Une opération à sémantique synchrone ayant pour objectif de soulever plusieurs exceptions successives devrait être décomposée en deux opérations SIR MHEG: l'une **synchronous** et l'autre **asynchronous**.

14.3.3 Paramètre implicite

Lorsqu'à une opération IDL correspondra une description de service de la SIR MHEG, l'instance d'objet auquel s'applique l'opération restera un paramètre implicite, à savoir il ne sera pas exprimé dans la signature du service.

NOTE – Cependant, sur invocation de l'opération, ce paramètre est fourni en tant que paramètre effectif de tête comme si son **type** était **object reference** et son **passing-mode** était **in**.

14.3.4 Valeur de retour

Au type de la valeur de retour d'une opération IDL correspondra le composant **return-value-type** de la description de service.

14.4 Attributs IDL

A un attribut IDL correspondront deux descriptions de service à l'intérieur d'une déclaration de bloc: un service "accessor" dont la fonction est d'extraire la valeur de l'attribut et un "modifier" dont la fonction est d'affecter la valeur de l'attribut.

14.4.1 Accesseur (Accessor)

Pour ce qui concerne le service "accessor", au nom global d'attribut IDL dont l'identificateur final est préfixé avec "get_" correspondra le composant de la SIR MHEG **name** de la description de service. Un service "accessor" ne possédera pas de paramètre explicite. Au type de l'attribut IDL correspondra le composant **return-value-type** de la description de service.

Exemple: dans l'API MHEG-3, à l'attribut **routine_id** de l'objet **RoutineInvocation** correspondra le nom global IDL **MHEG_3::RoutineInvocation::get_RoutineId**

14.4.2 Modificateur (Modifier)

Pour ce qui concerne le service "modifier", au nom global d'attribut IDL dont l'identificateur final est préfixé avec "set_" correspondra le composant de la SIR MHEG **name** de la description de service. Un service "modifier" possédera un paramètre avec un passing mode égal à **in** de telle sorte qu'au type d'attribut IDL correspondra le composant **type** de la description des paramètres de ce service. Un service "modifier" n'aura pas de valeur de retour.

14.4.3 Attribut en lecture seule

Si un attribut IDL est défini en tant que **readonly**, seul le service "accessor" sera fourni dans la déclaration de bloc.

14.5 Opérations IDL héritées

Le mappage des opérations IDL héritées s'effectuera comme si elles étaient définies dans l'interface spécifique.

14.6 Exceptions IDL

A une exception IDL correspondra un composant **exception-description** de la déclaration de bloc dans la SIR MHEG auquel mappe la spécification IDL à laquelle appartient l'exception.

14.6.1 Nom d'exception

Au nom global IDL d'une exception correspondra le composant **name** dans la SIR MHEG de cette description d'exception.

14.6.2 Membres d'une exception

Aux membres d'une exception IDL correspondra le composant **parameters-description** de cette description d'exception. Dans ce parameters-description, à chaque type de membre IDL correspondra le composant **type** identifiant un type déclaré conformément aux règles de mappage de type définies dans le présent paragraphe.

14.6.3 Membre implicite

Lorsqu'à une exception IDL correspondra une description d'exception de la SIR MHEG, l'instance d'objet de laquelle provient l'exception restera un membre implicite, à savoir elle ne sera pas exprimée dans la signature de l'exception.

NOTE – Cependant, lors de l'apparition de l'exception, ce membre est fourni en tant que membre effectif de tête comme si son **type** était "object reference".

14.7 Types IDL

A un type IDL correspondra un **TypeDeclaration** de la SIR MHEG déclaré comme un composant du composant **type-declarations** du script **InterchangedScript**. Une déclaration de type aura une portée globale au script échangé.

Aux types de base et aux constructeurs IDL correspondront des types primitifs et des constructeurs de la SIR MHEG. Ce mappage est résumé dans le Tableau 5:

Tableau 5/T.173 – Mappages de types

IDL	MHEG-SIR
void	void
octet	octet
short	short
unsigned short	unsigned short
long	long
unsigned long	unsigned long
float	float
double	double
boolean	boolean
char	character
enum	unsigned long
string	string
sequence	sequence
array	array
struct	structure
union	union
(object)	object reference
any	data identifier (voir ci-dessous)

14.7.1 Type **char**

Passer d'un type IDL **char** à un type **character** de la SIR MHEG donnera lieu à un transcodage de valeurs ISO 8859-1 vers les valeurs ISO/CEI 10646-1.

14.7.2 Type **enum**

La vérification de l'intervalle des valeurs **enum** n'a pas besoin d'être conservée.

14.7.3 Types construits

A une définition de type IDL correspondra un **TypeDescription** de la SIR MHEG. Si le type IDL est de base ou s'il a déjà été l'objet d'une autre déclaration de type, cette déclaration de type sera composée d'un identificateur de type. Dans le cas contraire, il sera construit conformément aux règles de mappage suivantes:

– à un champ IDL **struct** correspondra son rang dans la description SIR MHEG **structure**; son nom ne sera pas conservé;

– à une valeur d'étiquette IDL **union** correspondra son rang dans la description SIR MHEG **union**; son nom et sa valeur ne seront pas conservés;

– à un tableau multidimensionnel IDL **array** correspondra un tableau SIR MHEG **array** dont le type d'élément est **array**.

14.7.4 Type **any**

Au type IDL **any** correspondra le type **data identifier** de la SIR MHEG en supposant que le type **any** soit utilisé avec une clé associée pour déterminer le type effectif:

struct { Key the_key; any value }

où **Key** est un type **string**, **numeric** ou **enum** dont la valeur détermine complètement le type du champ **value**.

Au type IDL ci-dessus correspondra un type SIR MHEG **structure** composé de deux éléments:

– un **unsigned short** représentant un TID valide à l'intérieur du script pour mapper à la clé key;

– un **data identifier** représentant une variable de type identifiée par le premier élément et contenant la valeur value.

La conservation de la sémantique ne peut être garantie dans toute autre utilisation du type **any** lors du mappage vers la SIR MHEG.

14.7.5 Restrictions sur les types

Si deux types IDL ont la même structure, on leur fera correspondre un seul type SIR MHEG.

14.8 Constantes IDL

Aux constantes IDL correspondront un **ConstantDeclaration** SIR MHEG déclaré en tant que composant du composant **constant-declarations** du script **InterchangedScript**. Une déclaration de constante aura une portée globale au script échangé.

15 L'API MHEG-3

Le présent paragraphe spécifie la syntaxe et la sémantique de l'API MHEG-3.

Les scripts échangés utiliseront l'API MHEG-3 conformément à la syntaxe de l'interface IDL définie dans le présent paragraphe et dans l'Annexe F.

Les interpréteurs de script SIR MHEG fourniront l'API MHEG-3 conformément à la syntaxe de l'interface IDL définie dans le présent paragraphe et dans l'Annexe F, avec la sémantique définie dans le présent paragraphe. L'invocation des opérations aura l'effet spécifié dans le présent paragraphe.

Toutes les fonctions prédéfinies de la SIR MHEG auxquelles correspondent des opérations de l'API MHEG-3 seront **synchronous**.

La définition de l'API MHEG-3 est formée d'un module IDL unique appelé **MHEG_3**. Ce module définit les types prédéfinis, trois exceptions et quatre interfaces d'objet; il n'existe pas de relations d'héritage entre les quatre objets.

15.1 Objet interpréteur de script (ScriptInterpreter)

L'objet **ScriptInterpreter** représente l'interpréteur de script. Il sera unique. Il est utilisé pour fabriquer les objets **MhScript**.

Lorsqu'ils invoqueront des opérations sur l'objet **ScriptInterpreter**, les scripts échangés utiliseront la valeur "null" comme valeur du paramètre de la référence à objet implicite.

15.1.1 Opération Détruire (kill)

Synopsis

Interface:	**ScriptInterpreter**	
Opération:	**kill**	
Résultat:	**void**	

Description

L'opération **kill** est utilisée pur détruire l'objet **ScriptInterpreter** et pour terminer les processus d'interprétation du script.

Sur invocation de l'opération, le processus principal invoquera une opération destroy sur tous les objets **mh-scripts** disponibles et ensuite, arrêtera les processus d'interprétation de script.

A l'inverse des autres opérations de l'API MHEG-3, cette opération n'est pas une fonction SIR MHEG prédéfinie. Aussi, elle ne sera pas accessible aux scripts SIR MHEG échangés.

15.1.2 Opération Préparer (prepare)

Synopsis

Interface:	**ScriptInterpreter**	
Opération:	**prepare**	
Résultat:	**MhScript**	
In:	**ContentReference**	**content_reference**
Exception:	**InvalidParameter**	
Exception:	**InvalidScript**	
Exception:	**OperationFailed**	

Description

L'opération **prepare** est utilisée pour créer un objet **MhScript** à partir d'un script échangé et pour demander à l'interpréteur de script d'initialiser ce "mh-script".

Le paramètre **content_reference** spécifie l'emplacement du script échangé. Il est composé de deux chaînes: un identificateur public et un identificateur système. Si ces chaînes sont toutes les deux nulles, il sera ignoré. Au moins l'une des deux doit être non nulle.

Lors de l'invocation de l'opération, le processus principal traitera les opérations d'initialisation du "mh-script" conformément à ce qui est spécifié au 9.5.2. Une fois cette initialisation terminée, l'état du "mh-script" deviendra **available**.

Le résultat de l'opération sera une référence à objet en direction du **MhScript** créé.

L’exception **InvalidParameter** apparaîtra si le paramètre **content_reference** ne permet pas d’accéder à un script échangé. Ensuite le membre **rank** aura pour valeur 1.

L’exception **InvalidScript** apparaîtra si une déclaration illégale est détectée pendant l’analyse syntaxique du script échangé. Ensuite le membre **the_entity** représentera le type de la première entité dans la partie déclarations sur laquelle une erreur a été détectée, tandis que le membre **identifier** représentera l’identificateur de cette identité de la manière suivante:

– un TID pour une déclaration de type;

– un DID pour une déclaration de constante ou de variable;

– un FID pour une déclaration de service ou de routine;

– un MID pour une déclaration de filet ou d’exception;

– un PID pour une déclaration de bloc.

L’exception **OperationFailed** apparaîtra si les opérations d’initialisation du "mh-script" ne peuvent pas être terminées bien qu’aucune erreur n’ait été détectée dans la syntaxe du script échangé.

L’objet **MhScript** ne sera pas créé si une exception apparaît à n’importe quel moment et dans ce cas, l’état du "mh-script" restera à **not available**.

15.2 Objet **MhScript**

L’objet **MhScript** représente un "mh-script" "available". Il est utilisé pour fabriquer les objets **RtScript**.

15.2.1 Opération détruire (**destroy**)

Synopsis

Interface: **MhScript**

Opération: **destroy**

Résultat: **void**

Description

L’opération **destroy** est utilisée pour détruire l’objet **MhScript** ainsi que le "mh-script" correspondant.

Lors de l’invocation de l’opération, le processus principal effectuera les étapes suivantes dans cet ordre:

– mettre le "mh-script" destination à l’état **not available**;

– invoquer une opération **delete** sur tous les objets **RtScript** existants ayant été créés par ce "mh-script";

– effectuer la procédure **package unload** sur tous les blocs;

– libérer toutes les zones mémoire du "mh-script" qui lui sont affectées.

15.2.2 Opération new

Synopsis

Interface: **MhScript**

Opération: **new**

Résultat: **RtScript**

Exception: **OperationFailed**

Description

L'opération **new** est utilisée pour créer un objet **RtScript** à partir du "mh-script" et pour demander à l'interpréteur de script de l'initialiser.

Lors de l'invocation de l'opération, le processus principal traitera les opérations d'initialisation du "rt-script" comme il est spécifié au 9.5.3. Après une initialisation réussie, l'état du "rt-script" sera mis à **ready**.

Le résultat de l'opération sera une référence à objet sur le **RtScript** créé.

L'exception **OperationFailed** apparaîtra si les opérations d'initialisation du "rt-script" ne peuvent être terminées. Alors l'objet **RtScript** ne sera pas créé et l'état du "rt-script" restera à **not ready**.

15.3 Object RtScript

L'objet **RtScript** représente un "rt-script" dont l'état est soit **ready**, **running** ou **erroneous.** Il est utilisé pour fabriquer les objets **RoutineInvocation**.

15.3.1 Opération Détruire (delete)

Synopsis

Interface:	**RtScript**
Opération:	**delete**
Résultat:	**void**

Description

L'opération **delete** est utilisée pour détruire l'objet **RtScript** ainsi que le "rt-script" correspondant.

Lors de l'invocation de l'opération, le processus principal effectuera les étapes suivantes dans cet ordre:

– mettre le "rt-script" destination à l'état **not ready**;

– invoquer une opération **close** sur tous les objets **RoutineInvocation** ayant été créés par ce "rt-script";

– clore toutes les unités de traitement;

– libérer toutes les zones mémoire du "rt-script" qui lui sont affectées.

15.3.2 Opération Affectation de priorité (setPriority)

Synopsis

Interface:	**RtScript**	
Opération:	**setPriority**	
Résultat:	**void**	
In:	**unsigned short**	**priority**

Description

L'opération **setPriority** est utilisée pour modifier l'indication de priorité associée au "rt-script".

Le paramètre **priority** spécifie la nouvelle valeur de priorité.

Lors de l'invocation de l'opération, le processus principal peut modifier sa politique de priorité en conséquence. L'effet précis de cette opération n'est pas spécifié par la présente Recommandation. Sur certaines implémentations, il peut n'avoir aucun effet. Cependant, l'unité d'exécution d'un

"rt-script" ayant une valeur de priorité plus basse que celle d'un autre script ne devra pas recevoir plus de temps CPU que celle d'un autre.

15.3.3 Opération Extraction de priorité (**getPriority**)

Synopsis

Interface:	**RtScript**	
Opération:	**getPriority**	
Résultat:	**unsigned short**	

Description

L'opération **getPriority** est utilisée pour récupérer la valeur courante de l'indication de priorité associée au "rt-script". Si aucune priorité n'a été affectée à ce "rt-script" de manière explicite, la valeur par défaut spécifiée par l'interpréteur de script sera utilisée.

15.3.4 Opération Affectation de donnée (**setData**)

Synopsis

Interface:	**RtScript**	
Opération:	**setData**	
Résultat:	**void**	
In:	**DID**	**variable_id**
In:	**any**	**variable_value**
Exception:	**InvalidParameter**	
Exception:	**OperationFailed**	

Description

L'opération **setData** est utilisée pour affecter une valeur à une variable globale ou dynamique du "rt-script".

Le paramètre **variable_id** spécifie l'identificateur de donnée de la donnée à modifier.

Le paramètre **variable_value** spécifie la valeur devant être affectée à la variable. Le type du paramètre effectif est déterminé par le type de la variable.

Lors de l'invocation de l'opération, le processus principal demandera à l'unité d'exécution du "rt-script" d'affecter la valeur fournie à la variable destination.

L'exception **InvalidParameter** apparaîtra:

– si le paramètre **variable_id** référence une constante, une variable locale ou, une constante ou variable locale non existante. Dans ce cas, le membre **rank** sera mis à 1;

– Si le paramètre **variable_value** n'est pas d'un type IDL correspondant au type SIR MHEG de la variable destination. Dans ce cas, le membre **rank** sera mis à 2.

L'exception **OperationFailed** apparaîtra si l'état du "rt-script" est **running** ou **erroneous**.

15.3.5 Opération Extraction de donnée (**getData**)

Synopsis

Interface:	**RtScript**	
Opération:	**getData**	
Résultat:	**any**	
In:	**DID**	**data_id**

Exception: **InvalidParameter**

Exception: **OperationFailed**

Description

L'opération **getData** est utilisée pour récupérer la valeur courante d'une variable ou d'une constante.

Le paramètre **data_id** spécifie l'identificateur de donnée de la donnée à accéder.

Lors de l'invocation de l'opération, l'unité d'exécution du "rt-script" retournera la valeur courante de la constante ou de la variable.

Le résultat de l'opération sera la valeur demandée. Il sera d'un type IDL correspondant au type SIR MHEG de la constante ou variable destination.

L'exception **InvalidParameter** apparaîtra si le paramètre **data_id** référence une constante ou une variable inexistante. Dans ce cas, le membre **rank** sera mis à 1.

L'exception **OperationFailed** apparaîtra si l'état du "rt-script" est **running** ou **erroneous**.

15.3.6 Opération Allouer (Allocate)

Synopsis

Interface: **RtScript**

Opération: **allocate**

Résultat: **DID**

In: **TID** **variable_type_id**

Exception: **InvalidParameter**

Exception: **OperationFailed**

Description

L'opération **allocate** est utilisée pour créer une variable dynamique d'un type donné dans le "rt-script".

Le paramètre **variable_type_id** spécifie l'identificateur de type SIR MHEG de la variable destination selon sa déclaration dans le "rt-script".

Lors de l'invocation de l'opération, l'unité d'exécution du "rt-script" la traitera comme si elle exécutait une instruction **ALLOC** avec pour opérande **variable_type_id**, à savoir, elle réservera une mémoire de tas appropriée, générera un nouveau DID et le retournera.

Le résultat de l'opération sera l'identificateur de donnée de la nouvelle variable dynamique.

L'exception **InvalidParameter** apparaîtra si la valeur du paramètre **variable_type_id** n'est ni un type prédéfini, ni un type déclaré dans le "rt-script". Dans ce cas, le membre **rank** sera mis à 1.

L'exception **OperationFailed** apparaîtra à n'importe quel endroit où l'instruction **ALLOC** pourrait soulever une erreur. Dans ce cas, la variable ne sera pas allouée et le registre d'erreur ne sera pas modifié.

15.3.7 Opération Libérer (free)

Synopsis

Interface: **RtScript**

Opération: **free**

Résultat: **void**

In: **DID** **variable_id**

Exception: **InvalidParameter**

Description

L'opération **free** est utilisée pour détruire une variable dynamique du "rt-script".

Le paramètre **variable_id** spécifie l'identificateur de donnée de la variable à libérer.

Lors de l'invocation de l'opération, l'unité d'exécution du "rt-script" la traitera comme s'il exécutait une instruction **FREE** avec la paramètre **variable_id**, à savoir, il libérera la variable dynamique et invalidera son identificateur.

L'exception **InvalidParameter** apparaîtra si le paramètre **variable_id** ne référence pas une variable existante précédemment alloué à travers l'API MHEG-3. Dans ce cas, le membre **rank** sera mis à 1.

NOTE – Un interpréteur de script peut mettre en œuvre une politique d'allocation d'identificateurs de données permettant de distinguer facilement les variables allouées à travers l'API MHEG-3 des variables allouées en utilisant une instruction; par exemple cette distinction pourra se faire grâce à l'intervalle d'appartenance de leur identificateurs de données respectifs.

15.3.8 Opération Arrêter (stop)

Synopsis

Interface: **RtScript**

Opération: **stop**

Résultat: **void**

Exception: **OperationFailed**

Description

L'opération **stop** est utilisée pour remettre le "rt-script" à l'état "ready".

Lors de l'invocation de l'opération, l'interpréteur de script demandera à l'unité d'exécution de "rt-script", d'arrêter et ensuite de vider la pile d'appel, la file d'attente des messages et la pile des paramètres. Ensuite il mettra le "rt-script" à l'état **ready**. A l'opposé de l'opération **reInit** les variables globales et dynamiques resteront inchangées.

L'exception **OperationFailed** apparaîtra si l'opération ne peut pas se dérouler avec succès, par exemple en cas d'altération des zones mémoire du "rt-script" due à une erreur d'exécution.

15.3.9 Opération Réinitialisation (reInit)

Synopsis

Interface: **RtScript**

Opération: **reInit**

Résultat: **void**

Exception: **OperationFailed**

Description

L'opération **reInit** est utilisée pour remettre le "rt-script" à son état initial, à savoir juste après son initialisation.

Lors de l'invocation de l'opération, l'interpréteur de script:

– clora l'unité d'exécution de "rt-script";

– libérera les variables dynamiques;
– remettra les variables globales à leur état initial (comme dans la table de définition des variables globales du "mh-script");
– videra la pile des paramètres, la file d'attente des messages et la pile d'appel, libérant les tables de variables locales;
– réinitialisera tous les registres;
– enfin, mettra le "rt-script" à l'état **ready**.

L'exception **OperationFailed** apparaîtra si l'opération n'a pu être effectuée avec succès, par exemple en cas d'altération des zones mémoire du "rt-script" due à un erreur d'exécution.

15.3.10 Opération Extraction de statut de RtScript (getRtScriptStatus)

Synopsis

Interface:	**RtScript**	
Opération:	**getRtScriptStatus**	
Résultat:	**RtScriptStatus**	

Description

L'opération **getRtScriptStatus** est utilisée pour récupérer l'état courant du "rt-script".

Le résultat de l'opération sera l'un des états suivants: **READY**, **RUNNING** ou **ERRONEOUS**.

15.3.11 Opération Ouvrir (open)

Synopsis

Interface:	**RtScript**	
Opération:	**open**	
Résultat:	**RoutineInvocation**	
In:	**FID**	**routine_id**
Exception:	**InvalidParameter**	

Description

L'opération **open** est utilisée pour créer un objet **RoutineInvocation** à partir du "rt-script".

Le paramètre **routine_id** spécifie l'identificateur de fonction de la routine à laquelle est associé le nouvel objet **RoutineInvocation**.

L'interpréteur de script peut opter pour l'une des politiques suivantes:

a) récupérer la signature de la routine destination lors de l'invocation de l'opération **open**, de sorte que la validité des paramètres passés puisse être vérifiée au fil de l'eau, à savoir dès l'invocation d'une opération **setParameter**;

b) vérifier la validité des paramètres seulement lors de l'invocation de l'opération **run**.

Le résultat de l'opération sera une référence à objet vers l'objet **RoutineInvocation** créé.

L'exception **InvalidParameter** apparaîtra si le paramètre **routine_id** n'identifie pas une routine valide du "rt-script". Dans ce cas, le membre **rank** sera mis à 1.

15.4 Objet Invocation de routine (RoutineInvocation)

L'objet **RoutineInvocation** représente un contexte d'invocation de routine. Celui-ci est utilisé pour passer des paramètres et demander l'exécution d'une routine précise du "rt-script".

15.4.1 Opération Fermer (close)

Synopsis

Interface:	**RoutineInvocation**	
Opération:	**close**	
Résultat:	**void**	

Description

L'opération **close** est utilisée pour détruire l'objet **RoutineInvocation** et pour fermer le contexte correspondant d'invocation de la routine.

15.4.2 Attribut en lecture seule Identificateur de routine (routine_id)

Synopsis

Interface:	**RoutineInvocation**	
Attribute:	**FID**	**routine_id**

Description

L'attribut **routine_id** est de type lecture seule et est initialisé lors de la création de l'objet **RoutineInvocation** par l'opération **open**. Sa valeur sera l'identificateur de la fonction de la routine adressée par l'objet **RoutineInvocation.**

Les scripts échangés accéderont à la valeur de cet attribut en utilisant la fonction prédéfinie **get_RoutineId.**

15.4.3 Opération Affectation de paramètre (setParameter)

Synopsis

Interface:	**RoutineInvocation**	
Opération:	**setParameter**	
Résultat:	**void**	
In:	**unsigned short**	**rank**
In:	**TID**	**parameter_type_id**
In:	**any**	**parameter_value**
Exception:	**InvalidParameter**	

Description

L'opération **setParameter** est utilisée pour passer la valeur d'un paramètre à la routine en vue de son utilisation par la prochaine opération **run**.

Le paramètre **rank** spécifie le rang du paramètre passé, dans la description de signature de la routine, en commençant à partir de 0. Ainsi, celui-ci correspond à l'index du paramètre dans la table des variables locales de la routine.

Le paramètre **parameter_type_id** spécifie l'identificateur de type SIR MHEG du paramètre passé, déclaré à l'intérieur du script.

Le paramètre **parameter_value** spécifie la valeur du paramètre passé. Le type de la valeur est déterminé par le paramètre **parameter_type_id.**

Lors de l'invocation de l'opération, l'interpréteur de script bufferisera le paramètre en vue de son utilisation par l'opération **run** suivante appliquée à cette routine. Si l'interpréteur de script opte pour

la politique a) définie au 15.3.11, il vérifiera la validité des paramètres **parameter_type_id** et **parameter_value** par rapport à la signature de la routine.

Si l'interpréteur de script opte pour la politique a) définie au 15.3.11, l'exception **InvalidParameter** apparaîtra:

– si le paramètre **rank** de l'opération dépasse le numéro du dernier paramètre de la routine. Dans ce cas, le membre **rank** aura pour valeur 1;

– si le paramètre **parameter_type_id** ne correspond pas au type de paramètre dans la signature de la routine. Dans ce cas, le membre **rank** aura pour valeur 2;

– si le paramètre **parameter_value** n'est pas d'un type approprié, à savoir un type IDL correspondant au type décrit par le paramètre **parameter_type_id**, lorsque le mode de passage est par **value**, et à un type DID lorsque le mode de passage est par **reference**. Dans ce cas, le membre **rank** aura pour valeur 3;

– lorsque le mode de passage est par **reference**, si le paramètre **parameter_value** est un DID n'identifiant pas une variable globale ou dynamique existante dont le type correspond au type de paramètre défini par la signature de la routine. Dans ce cas, le membre **rank** aura pour valeur 3.

15.4.4 Opération Extraction de prototype (getPrototype)

Synopsis

Interface: **RoutineInvocation**

Opération: **getPrototype**

Résultat: **Prototype**

Description

L'opération **getPrototype** est utilisée pour récupérer la signature de la routine.

Lors de l'invocation de l'opération, l'interpréteur de script renverra la signature de la routine.

Le résultat de l'opération sera une description de la signature de la routine:

a) le champ **return_value_type** sera mis à RT[routine_id].TID;

b) le nième élément du champ signature correspondra à RT[routine_id].sig[n]:

1) le champ **passing-mode** sera mis respectivement à **BY_VALUE**, ou **BY_REFERENCE**, lorsque **RT[routine_id].sig[n].mod** est respectivement "value", "reference";

2) le champ **parameter_type_id** sera mis à **RT[routine_id].sig[n].TID**.

15.4.5 Opération Fonctionner (run)

Synopsis

Interface: **RoutineInvocation**

Opération: **run**

Résultat: **void**

Exception: **OperationFailed**

Description

L'opération **run** effectuera l'exécution de la routine avec les valeurs de paramètres précédemment fournies par l'opération **setParameter**.

Lors de l'invocation de l'opération le processus principal:

– créera un message dont l'identificateur de message est l'index de la routine (la valeur de l'attribut **routine_id)** et dont les paramètres sont les paramètres mis par les opérations précédentes **setParameter**;

– insérera ce message dans la file d'attente de messages du "rt-script" destination;

– si l'état courant du "rt-script" est **ready**, le mettre à **running**.

Si l'interpréteur de script opte pour la politique b) définie au 15.3.11, il vérifiera la validité, au regard de la signature de la routine, de tous les identificateurs de type et des valeurs des paramètres fournis précédemment par l'opération **setParameter**.

L'exception **OperationFailed** apparaîtra si n'importe lequel des paramètres fournis ne mappe pas à la signature de la routine.

15.4.6 Opération Remise à zéro (reset)

Synopsis

Interface: **RoutineInvocation**

Opération: **reset**

Résultat: **void**

Description

L'opération **reset** est utilisée pour vider le contexte d'invocation de la routine afin de préparer une nouvelle invocation.

Lors de l'invocation de l'opération, les paramètres précédemment bufferisés comme résultat d'une opération **setParameter** seront vidés.

NOTE – Le fait d'utiliser cette opération après chaque opération **run** évite tout risque de collision. Ne pas l'utiliser permet de répéter la même invocation sans fournir à nouveau les paramètres.

15.4.7 Opération Extraction du statut d'invocation (getInvocationStatus)

Synopsis:

Interface: **RoutineInvocation**

Opération: **getInvocationStatus**

Résultat: **InvocationStatus**

Description

L'opération **getInvocationStatus** est utilisée pour récupérer l'état d'invocation de la routine courante.

Le résultat de l'opération sera l'une des valeurs suivantes:

– **NOT_STARTED**: aucune opération **run** n'a été invoquée depuis la création de l'objet ou depuis la dernière opération **reset**;

– **PROCESSING**: une opération **run** a été invoquée mais l'exécution de la routine par l'unité d'exécution de "rt-script" n'est pas terminée (soit la demande est dans la file d'attente de messages ou la routine est en cours d'exécution);

– **TERMINATED**: l'exécution de la routine déclenchée par la dernière opération **run** invoquée a été terminée par l'unité d'exécution de *rt*-script;

– **ABORTED**: l'exécution de la routine déclenchée par la dernière opération run invoquée a abouti sur une erreur d'exécution d'instruction.

ANNEXE A

Spécification ASN.1 des scripts échangés

La présente annexe spécifie la notation ASN.1 pour la syntaxe du composant " données de script " de la classe MHEG "script" conformément à la Rec. UIT-T X.680 | ISO/CEI 8824-1 [1].

Les scripts échangés auront la syntaxe définie dans le module ASN.1 **ISOMHEG-sir**.

```
-- Module: MHEG-SIR (sir)--
--
-- Copyright statement:
-- --------------------
--      (c) ITU, 1996.
--          Permission to copy in any form is granted for use with conforming to
--          MHEG-3 engines and applications as defined by this Recommendation
--          provided this notice is included in all copies.

ISOMHEG-sir {joint-iso-itu-t (2) mheg (19) version (1) script-interchange-representation (11)}
DEFINITIONS IMPLICIT TAGS ::= BEGIN

EXPORTS    InterchangedScript;

InterchangedScript          ::=     SEQUENCE
{
        type-declarations                   SEQUENCE (SIZE (1.. max-nb-declared-types)) OF
                                    TypeDeclaration OPTIONAL,
        constant-declarations               [0] SEQUENCE (SIZE (1 .. max-nb-constants)) OF
                                    ConstantDeclaration OPTIONAL,
        global-variable-declarations        [1] SEQUENCE (SIZE (1 .. max-nb-global-variables)) OF
                                    VariableDeclaration OPTIONAL,
        external-package-declarations       [2] SEQUENCE (SIZE (1 .. max-nb-packages)) OF
                                    PackageDeclaration OPTIONAL,
        handler-declarations                [3] SEQUENCE (SIZE (1 .. max-nb-messages)) OF
                                    HandlerDeclaration OPTIONAL,
        routine-declarations                [4] SEQUENCE (SIZE (1 .. max-nb-routines)) OF
                                    RoutineDeclaration OPTIONAL
}

TypeDeclaration     ::=     SEQUENCE
{
        identifier                  [0] TypeIdentifier OPTIONAL,
        description                 TypeDescription
}

TypeDescription     ::=     CHOICE
{
        string-description              [1] INTEGER (0..max-size-string) OPTIONAL,
        sequence-description            [2] SequenceDescription,
        array-description               [3] ArrayDescription,
        structure-description           [4] StructureDescription,
        union-description               [5] UnionDescription
}

SequenceDescription         ::=     SEQUENCE
{
        bound                       INTEGER (0 .. max-size-sequence),
        element-type                TypeIdentifier
}
```

```
ArrayDescription   ::=   SEQUENCE
{
        size                    INTEGER (1 .. max-size-array),
        element-type            TypeIdentifier
}

UnionDescription   ::=   SEQUENCE (SIZE (1 .. max-size-union)) OF TypeIdentifier

StructureDescription    ::=   SEQUENCE (SIZE (1 .. max-size-structure)) OF TypeIdentifier

ConstantDeclaration     ::=   SEQUENCE
{
        identifier              [0] DataIdentifier OPTIONAL,
        type                    TypeIdentifier ALL EXCEPT 0,
        value                   ConstantValue
}
ConstantValue           ::=   CHOICE
{
        octet                           [1] OctetValue,
        short                           [2] ShortValue,
        long                            [3] LongValue,
        unsigned-short                  [4] UnsignedShortValue,
        unsigned-long                   [5] UnsignedLongValue,
        float                           [6] FloatValue,
        double                          [7] DoubleValue,
        boolean                         [8] BooleanValue,
        character                       [9] CharacterValue,
        data-identifier                 [10] DataIdentifier (0..<max-nb-constants),
        string                          [11] StringValue,
        sequence                        [12] SequenceValue,
        array                           [13] ArrayValue,
        structure                       [14] StructureValue,
        union                           [15] UnionValue
}

SequenceValue           ::=   SEQUENCE (SIZE (0 .. max-size-sequence)) OF ConstantValue

ArrayValue        ::=   SEQUENCE (SIZE (1 .. max-size-array)) OF ConstantValue

UnionValue        ::=   SEQUENCE
{
        tag                     INTEGER (0 .. < max-size-union),
        value                   ConstantValue
}

StructureValue          ::=   SEQUENCE (SIZE (1 .. max-size-structure)) OF ConstantValue

VariableDeclaration     ::=   SEQUENCE
{
        identifier                      [0] DataIdentifier OPTIONAL,
        type                            TypeIdentifier,
        initial-value                   ConstantReference OPTIONAL
}

PackageDeclaration      ::=   SEQUENCE
{
        identifier              [0] PackageIdentifier OPTIONAL,
        name                    VisibleString OPTIONAL,
        services                SEQUENCE (SIZE (0 .. max-nb-services)) OF
                                        ServiceDescription,
        exceptions              SEQUENCE (SIZE (0 .. max-nb-exceptions)) OF
```

```
}

ServiceDescription ::=     SEQUENCE
{
        identifier                      [0] FunctionIdentifier OPTIONAL,
        name                            VisibleString OPTIONAL,
        calling-mode                    ENUMERATED {synchronous (0), asynchronous (1)}
                                            DEFAULT synchronous,
        return-value-type               TypeIdentifier DEFAULT 0,
        parameters-description          SEQUENCE OF ServiceParameterDescription OPTIONAL
}

ServiceParameterDescription     ::=     SEQUENCE
{
        passing-mode                    ENUMERATED {in (1), out (2), inout (3)} DEFAULT in,
        type                            TypeIdentifier ALL EXCEPT 0
}

ExceptionDescription        ::=     SEQUENCE
{
        identifier                          [0] MessageIdentifier OPTIONAL,
        name                                VisibleString OPTIONAL,
        parameters-description              SEQUENCE OF TypeIdentifier OPTIONAL
}

HandlerDeclaration          ::=     SEQUENCE
{
        message-identifier              MessageIdentifier,
        function-identifier             FunctionIdentifier
}

RoutineDeclaration ::=     SEQUENCE
{
        routine-description             RoutineDescription,
        program-code                    OCTET STRING
}

RoutineDescription ::=     SEQUENCE
{
        identifier                          [0] FunctionIdentifier OPTIONAL,
        return-value-type                   TypeIdentifier DEFAULT 0,
        parameters-description              [1] SEQUENCE OF RoutineParameterDescription OPTIONAL,
        local-variable-table                [2] SEQUENCE (SIZE (0 .. max-nb-local-variables)) OF
                                                VariableDeclaration OPTIONAL
}

RoutineParameterDescription     ::= SEQUENCE
{
        passing-mode                    ENUMERATED {value (1), reference (3)} DEFAULT value,
        type                            TypeIdentifier ALL EXCEPT 0
}

ConstantReference ::=     CHOICE
{
        identifier              [16] DataIdentifier,
        value                   ConstantValue
}

max-size-sequence               INTEGER   ::=   65535
max-size-string                 INTEGER   ::=   65535
```

```
max-size-array                  INTEGER  ::=  65536
max-size-union                  INTEGER  ::=  256
max-size-structure              INTEGER  ::=  256
max-nb-global-variables         INTEGER  ::=  28672
max-nb-constants                INTEGER  ::=  4096
max-nb-local-variables          INTEGER  ::=  256
max-nb-dynamic-variables        INTEGER  ::=  32512
max-nb-data                     INTEGER  ::=  65536
    -- max-nb-constants+max-nb-global-variables+max-nb-local-variables+max-nb-dynamic-
    -- variables
max-nb-packages                 INTEGER  ::=  192
max-nb-services                 INTEGER  ::=  256
max-nb-routines                 INTEGER  ::=  4096
max-nb-predef-functions         INTEGER  ::=  12288
max-nb-functions                INTEGER  ::=  65536
    -- max-nb-packagesxmax-nb-services+max-nb-predef-functions+max-nb-routines
max-nb-exceptions               INTEGER  ::=  256
max-nb-predef-messages          INTEGER  ::=  16384
max-nb-messages                 INTEGER  ::=  65536
    -- max-nb-packagesxmax-nb-exceptions+max-nb-predef-messages
max-nb-declared-types           INTEGER  ::=  16384
max-nb-predef-types             INTEGER  ::=  16384
max-nb-types                    INTEGER  ::=  32768
    -- max-nb-predef-types + max-nb-declared-types

OctetValue                      ::=  OCTET STRING (SIZE (1))
ShortValue                      ::=  INTEGER (–32768 .. 32767)
LongValue                       ::=  INTEGER (–2147483648 .. 2147483647)
UnsignedShortValue              ::=  INTEGER (0 .. 65535)
UnsignedLongValue               ::=  INTEGER (0 .. 4294967295)
FloatValue                      ::=  REAL
DoubleValue                     ::=  REAL
BooleanValue                    ::=  BOOLEAN
CharacterValue                  ::=  BMPString (SIZE (1))
StringValue                     ::=  BMPString (SIZE (0.. max-size-string))

TypeIdentifier                  ::=  INTEGER (0 .. < max-nb-types)
DataIdentifier                  ::=  INTEGER (0 .. < max-nb-data)
FunctionIdentifier              ::=  INTEGER (0 .. < max-nb-functions)
MessageIdentifier               ::=  INTEGER (0 .. < max-nb-messages)
PackageIdentifier               ::=  INTEGER (0 .. < max-nb-packages)

END
```

ANNEXE B

Représentation codée des scripts échangés

B.1 Codage des scripts échangés

Les scripts échangés seront codés selon les règles de codage distinctes ASN.1 (DER, *distinguished coding rules*) spécifiées par la Rec. UIT-T X.690 | ISO/CEI 8825-1 [2].

NOTE – Ceci a pour but de rendre la tâche de décodage du moteur MHEG-3 aussi efficace que possible en enlevant toutes les options de codage ASN.1 pouvant l'allonger ou la compliquer.

B.2 Codage du code programme

La valeur du composant **program-code** du type **RoutineDeclaration** défini par **ISOMHEG-sir** (voir Annexe A) sera codée selon les règles définies dans le présent sous-paragraphe.

La séquence d'instructions fabriquant le code programme d'une routine sera codée comme une séquence d'octets. L'ordre d'encodage y sera le même que celui de l'ordre attendu d'exécution des instructions.

Chaque instruction sera codée en utilisant un octet pour le code opérateur, suivi de zéro à trois octets pour les opérandes. Ce dernier nombre dépend du code opérateur.

B.2.1 Instructions code opérateur

Les codes opérateurs seront codés en utilisant la chaîne de bits définie dans le Tableau B.1.

B.2.2 Instructions opérande

Les opérandes auront la longueur et le codage définis dans le Tableau B.1 selon le code opérateur de l'instruction. Tous les opérandes tenant sur plusieurs octets seront codés selon un ordre poids forts – poids faibles.

B.2.2.1 Opérandes Data identifier (identificateur de donnée)

Les opérandes DID seront codés sur deux octets comme suit:

– lorsque le bit 16 est à '1' et les bits 15 à 9 sont à '0', le DID référencera une variable locale, les bits 8 à 1 représentant l'index de la variable locale (de 0 à 255);

– dans les autres cas où le bit 16 est à 1, le DID référencera une variable dynamique, les bits 15 à 1 représentant l'index de variable dynamique (de 0 à 32511) incrémenté de 256.

– lorsque la valeur des bits 16 à 13 est '0000', le DID référencera une constante, les bits 12 à 1 représentant l'index de la constante (de 0 à 4095);

– dans tous les autres cas, le DID référencera une variable globale, les bits 15 à 1 représentant l'index de la variable globale (de 0 à 28671) incrémenté de 4096.

B.2.2.2 Opérandes Function identifier (identificateur de fonction)

Les opérandes FID seront encodés sur deux octets comme suit:

– lorsque la valeur des bits 16 à 13 est '0000', le FID référencera une routine, les bits 12 à 1 représentant l'index de la routine (de 0 à 4095);

– dans les autres cas où les bits 16 et 15 valent '00', le FID référencera une fonction prédéfinie, les bits 14 à 1 représentant l'index de la fonction prédéfinie (de 0 à 12287) incrémenté de 4096;

– dans tous les autres cas, le FID référencera un service, les bits 16 à 9 représentant l'identificateur de bloc (de 0 à 191) incrémenté de 64, et les bits 8 à 1 représentant l'index du service (de 0 à 255) dans ce bloc.

B.2.2.3 Opérandes numériques divers

Les opérandes ayant un " offset " sur un octet seront codés en complément à 1 sur 1 octet, le bit 8 représentant la direction du décalage, les bits 7 à 1 représentant le nombre d'unités de décalage dans la direction.

Les opérandes ayant un " offset " sur deux octets seront codés en complément à 1 sur 2 octets, le bit 16 représentant la direction du décalage, les bits 15 à 1 représentant le nombre d'unités de décalage dans la direction.

Les opérandes de type *value* seront codés en complément à deux sur deux octets pour être interprétés comme des valeur entières signées.

Les opérandes de type *index* seront codés sur un octet pour être interprétés comme des valeurs entières non signées.

Tableau B.1/T.173 – Codage des instructions de la SIR MHEG

Mnémoniques d'instructions	Code opérateur (binaire)	Code opérateur (hexa)	Long. Op1	Codage de l'Op1	Long. Op2	Codage de l'Op2
NOP	0000 0000	00	0			
YIELD	0000 0010	02	0			
RET	0000 0011	03	0			
FREE	0000 1000	08	0			
NOT_B	0001 0000	10	0			
NOT_O	0001 0001	11	0			
NOT_W	0001 0010	12	0			
NOT_U	0001 0011	13	0			
OR_B	0001 0100	14	0			
OR_O	0001 0101	15	0			
OR_W	0001 0110	16	0			
OR_U	0001 0111	17	0			
XOR_B	0001 1000	18	0			
XOR_O	0001 1001	19	0			
XOR_W	0001 1010	1A	0			
XOR_U	0001 1011	1B	0			
AND_B	0001 1100	1C	0			
AND_O	0001 1101	1D	0			
AND_W	0001 1110	1E	0			
AND_U	0001 1111	1F	0			
EQR	0010 0000	20	0			
EQ_O	0010 0001	21	0			
EQ_S	0010 0010	22	0			
EQ_L	0010 0011	23	0			
EQ_W	0010 0100	24	0			
EQ_U	0010 0101	25	0			
EQ_F	0010 0110	26	0			
EQ_D	0010 0111	27	0			
EQ_B	0010 1000	28	0			
EQ_C	0010 1001	29	0			
EQ_I	0010 1010	2A	0			
EQ_R	0010 1011	2B	0			
LT_C	0011 0000	30	0			
LT_O	0011 0001	31	0			
LT_S	0011 0010	32	0			
LT_L	0011 0011	33	0			

Tableau B.1/T.173 – Codage des instructions de la SIR MHEG *(suite)*

Mnémoniques d'instructions	Code opérateur (binaire)	Code opérateur (hexa)	Long. Op1	Codage de l'Op1	Long. Op2	Codage de l'Op2
LT_W	0011 0100	34	0			
LT_U	0011 0101	35	0			
LT_F	0011 0110	36	0			
LT_D	0011 0111	37	0			
GT_C	0011 1000	38	0			
GT_O	0011 1001	39	0			
GT_S	0011 1010	3A	0			
GT_L	0011 1011	3B	0			
GT_W	0011 1100	3C	0			
GT_U	0011 1101	3D	0			
GT_F	0011 1110	3E	0			
GT_D	0011 1111	3F	0			
ADD_O	0100 0001	41	0			
ADD_S	0100 0010	42	0			
ADD_L	0100 0011	43	0			
ADD_W	0100 0100	44	0			
ADD_U	0100 0101	45	0			
ADD_F	0100 0110	46	0			
ADD_D	0100 0111	47	0			
SUB_O	0100 1001	49	0			
SUB_S	0100 1010	4A	0			
SUB_L	0100 1011	4B	0			
SUB_W	0100 1100	4C	0			
SUB_U	0100 1101	4D	0			
SUB_F	0100 1110	4E	0			
SUB_D	0100 1111	4F	0			
MUL_O	0101 0001	51	0			
MUL_S	0101 0010	52	0			
MUL_L	0101 0011	53	0			
MUL_W	0101 0100	54	0			
MUL_U	0101 0101	55	0			
MUL_F	0101 0110	56	0			
MUL_D	0101 0111	57	0			
DIV_O	0101 1001	59	0			
DIV_S	0101 1010	5A	0			
DIV_L	0101 1011	5B	0			
DIV_W	0101 1100	5C	0			
DIV_U	0101 1101	5D	0			
DIV_F	0101 1110	5E	0			
DIV_D	0101 1111	5F	0			

Tableau B.1/T.173 – Codage des instructions de la SIR MHEG *(suite)*

Mnémoniques d'instructions	Code opérateur (binaire)	Code opérateur (hexa)	Long. Op1	Codage de l'Op1	Long. Op2	Codage de l'Op2
NEG_S	0110 0010	62	0			
NEG_L	0110 0011	63	0			
NEG_F	0110 0110	66	0			
NEG_D	0110 0111	67	0			
REM_O	0111 1001	79	0			
REM_S	0111 1010	7A	0			
REM_L	0111 1011	7B	0			
REM_W	0111 1100	7C	0			
REM_U	0111 1101	7D	0			
DUP_O	1000 0001	81	0			
DUP_S	1000 0010	82	0			
DUP_L	1000 0011	83	0			
DUP_W	1000 0100	84	0			
DUP_U	1000 0101	85	0			
DUP_F	1000 0110	86	0			
DUP_D	1000 0111	87	0			
DUP_B	1000 1000	88	0			
DUP_C	1000 1001	89	0			
DUP_I	1000 1010	8A	0			
DUP_R	1000 1011	8B	0			
CVT_SW	1001 0100	94	0			
CVT_WS	1001 0101	95	0			
CVT_LU	1001 0110	96	0			
CVT_UL	1001 0111	97	0			
CVT_CW	1001 1010	9A	0			
CVT_WC	1001 1011	9B	0			
CVT_BS	1010 0000	A0	0			
CVT_OS	1010 0001	A1	0			
CVT_SL	1010 0010	A2	0			
CVT_LF	1010 0011	A3	0			
CVT_WL	1010 0100	A4	0			
CVT_UF	1010 0101	A5	0			
CVT_FD	1010 0110	A6	0			
CVT_BO	1010 1000	A8	0			
CVT_OW	1010 1001	A9	0			
CVT_SU	1010 1010	AA	0			
CVT_WU	1010 1100	AC	0			
CVT_OB	1011 0001	B1	0			
CVT_SB	1011 0010	B2	0			
CVT_LB	1011 0011	B3	0			

Tableau B.1/T.173 – Codage des instructions de la SIR MHEG *(suite)*

Mnémoniques d'instructions	Code opérateur (binaire)	Code opérateur (hexa)	Long. Op1	Codage de l'Op1	Long. Op2	Codage de l'Op2
CVT_WB	1011 0100	B4	0			
CVT_UB	1011 0101	B5	0			
CVT_WO	1011 1001	B9	0			
CVT_LS	1011 1010	BA	0			
CVT_FL	1011 1011	BB	0			
CVT_UW	1011 1100	BC	0			
CVT_FU	1011 1101	BD	0			
CVT_DF	1011 1110	BE	0			
JT	1100 0000	C0	1	offset(signé)		
JF	1100 0001	C1	1	offset(signé)		
JMP	1100 0010	C2	1	offset(signé)		
SHIFT_O	1100 0101	C5	1	offset(signé)		
SHIFT_W	1100 0110	C6	1	offset(signé)		
SHIFT_U	1100 0111	C7	1	offset(signé)		
GETOR	1100 1001	C9	1	identificateur de bloc		
LJT	1101 0000	D0	2	offset(signé)		
LJF	1101 0001	D1	2	offset(signé)		
LJMP	1101 0010	D2	2	offset(signé)		
CALL	1101 0100	D4	2	identificateur de fonction		
XCALL	1101 0110	D6	2	identificateur de fonction		
PUSH	1110 0000	E0	2	identificateur de fonction		
PUSHR	1110 0001	E1	2	identificateur de fonction		
PUSHI	1110 0011	E3	2	valeur(signée)		
POP	1110 0100	E4	2	identificateur de donnée		
POPR	1110 0101	E5	2	identificateur de donnée		
POPC	1110 0110	E6	2	identificateur de donnée		
ALLOC	1110 1000	E8	2	identificateur de type		
INC	1110 1100	EA	2	identificateur de donnée		
DEC	1110 1101	EB	2	identificateur de donnée		
GET	1111 0000	F0	2	identificateur de donnée	1	index (non signé)

Tableau B.1/T.173 – Codage des instructions de la SIR MHEG *(fin)*

Mnémoniques d'instructions	Code opérateur (binaire)	Code opérateur (hexa)	Long. Op1	Codage de l'Op1	Long. Op2	Codage de l'Op2
GETC	1111 0010	F2	2	identificateur de donnée	1	index (non signé)
SET	1111 0100	F4	2	identificateur de donnée	1	index (non signé)
SETC	1111 0110	F6	2	identificateur de donnée	1	index (non signé)

ANNEXE C

Eléments prédéfinis de la SIR MHEG

La présente annexe donne la liste des messages, types et fonctions prédéfinis de la SIR MHEG avec leurs indices correspondants.

Les messages, types et fonctions prédéfinis peuvent être référencés par leur identificateur et utilisés dans les scripts échangés d'une manière identique à ce qu'il en serait pour les types, messages et fonctions déclarés dans la partie des déclarations globales des scripts échangés.

C.1 Types prédéfinis

Les types prédéfinis de la SIR MHEG comprennent

– des types primitifs;

– des types API MHEG.

C.1.1 Types primitifs

Les types primitifs définis dans la présente Recommandation seront codés en utilisant les identificateurs de types prédéfinis dont la liste se trouve dans le Tableau C.1.

Tableau C.1/T.173 – Identificateurs de types prédéfinis correspondant aux types primitifs

Nom du type	Identificateur de type
vide **void**	0
octet **octet**	1
entier court **short**	2
entier long **long**	3
entier court non signé **unsigned short**	4
entier long non signé **unsigned long**	5
flottant **float**	6
double **double**	7
booléen **boolean**	8

Tableau C.1/T.173 – Identificateurs de types prédéfinis correspondant aux types primitifs *(fin)*

Nom du type	Identificateur de type
caractère **character**	9
identificateur de donnée **data identifier**	10
référence d'objet **object reference**	11

On peut construire tous les types exprimables de la SIR MHEG (y compris les types MHEG prédéfinis) avec les types primitifs de la SIR MHEG et les constructeurs suivants:

– chaîne **string**;

– séquence **sequence**;

– matrice **array**;

– structure **structure**;

– union **union**.

Par convention, le type **string** non lié (le seul type construit sans élément ni paramètre) sera prédéfini avec un identificateur de type égal à 12.

C.1.2 Types de l'API MHEG

Les types de l'API MHEG définis par l'API MHEG seront codés en utilisant des identificateurs de types prédéfinis.

NOTE – Les types de l'API MHEG ont pour objectif d'être utilisés par les scripts échangés pour exprimer des informations échangées entre l'interpréteur de script et les entités MHEG.

On fera correspondre aux descriptions des types de la SIR MHEG la définition IDL de ces types fournie par l'API MHEG, en utilisant les règles générales de mappage IDL définies dans le paragraphe 14 et les règles spécifiques de mappage de l'API MHEG définies au E.2.

C.2 Fonctions prédéfinies

Les fonctions prédéfinies de la SIR MHEG comprennent

– les opérations de l'API MHEG;

– les opérations de l'API MHEG-3.

C.2.1 Opérations de l'API MHEG

Les opérations de l'API MHEG définies par l'API MHEG seront codées en utilisant les identificateurs de fonctions prédéfinis.

Les identificateurs de message prédéfinis des opérations de l'API MHEG commenceront à la valeur 1100h.

On fera correspondre aux descriptions des fonctions de la SIR MHEG la définition IDL de ces opérations fournie par l'API MHEG-3, en utilisant les règles générales de mappage IDL définies dans le paragraphe 14 et les règles spécifiques de mappage de l'API MHEG définies au E.2.

C.2.2 Opérations de l'API MHEG-3

Les opérations de l'API MHEG-3 définies par l'API MHEG-3 (voir le paragraphe 15) seront codées en utilisant les identificateurs des fonctions prédéfinies selon le Tableau C.2.

Tableau C.2/T.173 – Les identificateurs des fonctions prédéfinies correspondant aux opérations API MHEG-3

Operation name	Predefined function index	Function identifier
prepare	0	1000h
destroy	1	1001h
new	2	1002h
delete	3	1003h
setPriority	4	1004h
getPriority	5	1005h
setData	6	1006h
getData	7	1007h
allocate	8	1008h
free	9	1009h
stop	10	100Ah
reInit	11	100Bh
getRtScriptStatus	12	100Ch
open	13	100Dh
close	14	100Eh
getRoutineId	15	100Fh
setParameter	16	1010h
getPrototype	17	1011h
run	18	1012h
reset	19	1013h
getInvocationStatus	20	1014h

On fera correspondre les descriptions de fonctions de la SIR MHEG aux définitions IDL de ces opérations (voir Annexe F) en utilisant les règles de mappage IDL définies au 14.

C.3 Messages prédéfinis

Les messages prédéfinis de la SIR MHEG à destination d'un "rt-script" résultent:

– de l'invocation de l'opération **run** de l'API MHEG-3;

– de l'exception **InstructionExecutionError**;

– des exceptions de l'API MHEG-3;

– des exceptions de l'API MHEG.

C.3.1 Opérations de l'API MHEG-3

L'identificateur du message résultant de l'invocation d'une opération **run**, selon la définition du 15.4.5, sera égal à l'identificateur de fonction de la routine destination.

Les messages résultant d'opérations de l'API MHEG-3 auront alors un identificateur de message ayant une valeur comprise entre 0 et 0FFFh.

C.3.2 L'exception InstructionExecutionError (erreur d'exécution d'instruction)

L'exception **InstructionExecutionError**, selon la définition du 9.5.2, aura 1000h pour valeur d'identificateur de message.

L'exception **InstructionExecutionError** aura un membre de type **unsigned long**, dont la valeur sera mise à celle de ER.

Le code majeur d'erreur déterminera l'octet le moins significatif du membre (et le registre ER) selon la définition du Tableau C.3.

Tableau C.3/T.173 – Codes d'erreur d'exécution d'instruction

Error name	Error code
InvalidOperand	1
InvalidParameter	2
InvalidType	3
InvalidIdentifier	4
InvalidLevel	5
InvalidIndex	6
StackUnderflow	7
ArithmeticOverflow	8
DivisionByZero	9
HandlerNotFound	10
InvalidReturnValue	11
BadPackageStatus	12
InvalidObjectReference	13
TypeMismatch	14
JumpOutOfRange	15
AllocationFailed	16

C.3.3 Exceptions de l'API MHEG-3

Les exceptions de l'API MHEG-3 définies au paragraphe 15, auront les identificateurs de messages définis dans le Tableau C.4.

Tableau C.4/T.173 – Identificateurs de messages prédéfinis pour les exceptions de l'API MHEG-3

Nom de l'exception	Index de message prédéfini	Identificateur de message
InvalidScript	1	1001h
InvalidParameter	2	1002h
OperationFailed	3	1003h

Aux définitions IDL de ces exceptions définies dans l'Annexe F, correspondront des descriptions de message SIR MHEG utilisant les règles de mappage IDL définies dans le paragraphe 14.

C.3.4 Exceptions de l'API MHEG

Les exceptions de l'API MHEG définies dans l'API MHEG seront codées en utilisant des identificateurs de messages prédéfinis.

Les identificateurs de messages prédéfinis pour les exceptions de l'API MHEG commenceront à partir de 1100h.

Aux définitions IDL de ces exceptions fournies par l'API MHEG correspondront des descriptions de messages utilisant les règles générales de mappage définies au paragraphe 14 et les règles spécifiques de mappage de l'API MHEG définies au E.2.

ANNEXE D

Formulaire de spécification de mappage de plate-forme IDL

Les moteurs MHEG-3 permettront d'accéder aux services fournis par l'environnement d'exécution d'une plate-forme donnée, en supposant que cet environnement soit conforme à la spécification de mappage de plate-forme enregistrée pour cette plate-forme.

La spécification de mappage de plate-forme enregistrée sera fournie conformément au modèle spécifié dans la présente annexe, avec tous les champs dûment complétés.

Cette spécification de mappage de plate-forme de la SIR MHEG définit les mécanismes que les moteurs MHEG-3 auront besoin d'utiliser pour accéder aux services fournis par l'environnement d'exécution sur cette plate-forme.

Procédure de **description de plate-forme**

La plate-forme à laquelle s'applique cette spécification est <platform_description>.

Procédure de **disponibilité de bloc**

Pour savoir si une spécification IDL est disponible dans un environnement d'exécution et pour la localiser, un moteur MHEG-3 procédera selon la procédure <package_availability_procedure>

Procédure de **chargement de bloc**

Pour rendre accessible les opérations d'une spécification IDL disponible, un moteur MHEG-3 procédera selon la procédure <package_load_procedure>

Procédure de **déchargement de bloc**

Pour arrêter les opérations accessibles d'une spécification IDL disponible, un moteur MHEG-3 procédera selon la procédure. <package_unload_procedure>

Procédure de **invocation d'opération**

Pour invoquer une opération d'une spécification IDL accessible, un moteur MHEG-3 procédera selon la procédure. <operation_invocation_procedure>

Procédure de **passage de paramètre**

Lors de l'invocation d'une opération IDL, un moteur MHEG-3 passera les paramètres **in** à l'aide de la procédure. <in_parameter_passing_procedure>

Lors de l'invocation d'une opération IDL, un moteur MHEG-3 passera les paramètres **out** à l'aide de la procédure. <out_parameter_passing_procedure>

Lors de l'invocation d'une opération IDL, un moteur MHEG-3 passera les paramètres **inout** à l'aide de la procédure. <inout_parameter_passing_procedure>

Procédure de **récupération de paramètre de sortie**

Pour récupérer les valeurs des paramètres **in** ou **inout** à la suite de l'invocation d'une opération IDL, un moteur MHEG-3 procédera selon la procédure. <output_parameter_retrieval_procedure>

Procédure de **récupération de valeur de retour**

Pour récupérer la valeur de retour d'une opération IDL précédemment invoquée, un moteur MHEG-3 procédera selon la procédure. <return_value_retrieval_procedure>

Règles de codage de données

Les valeurs des données échangées entre le moteur MHEG-3 et l'environnement d'exécution seront codées selon les règles. <data_encoding_rules>

Procédure de **récupération d'exception**

Pour récupérer les exceptions qui apparaissent dans l'environnement d'exécution, un moteur MHEG-3 procédera selon la procédure. <exception_retrieval_procedure>

Exceptions système

Les exceptions système pouvant apparaître dans l'environnement d'exécution et pouvant être récupérées par un moteur MHEG-3 sont définies selon les définitions <system_exception_definitions>

Limitations de ressources

Lors de l'utilisation d'un environnement d'exécution sur la plate-forme, les limitations de ressources qui s'appliquent sont données par <resource_limitations_statement>

ANNEXE E

Processus de définition de l'API MHEG

La présente Recommandation générique ne définit pas une API MHEG spécifique (voir 8.3.3). Elle définit à la place un ensemble de règles et de procédures génériques applicables à la définition de l'API MHEG devant être fournie par toute Recommandation décrivant des objets de présentation. Ceci comprend:

– les règles qui seront utilisées pour produire la définition de l'API MHEG (voir E.1);
– la procédure qui sera utilisée pour définir le mappage SIR MHEG de cette API MHEG (voir E.2).

E.1 Cadre de définition de l'API générique

Le fait de produire une spécification de l'API MHEG à partir d'une autre Recommandation décrivant des objets de présentation (appelés ci-après spécification MHEG) est un processus qui consiste à produire des éléments IDL à partir des éléments MHEG.

Les éléments MHEG auxquels s'applique ce processus sont décrits au E.1.1. Les éléments IDL à produire à partir de ces éléments MHEG sont décrits au E.1.2. Les règles utilisées pour produire les éléments IDL à partir des éléments MHEG sont décrites dans les sous-paragraphes E.1.3 et suivants.

E.1.1 Eléments MHEG en entrée du processus de définition de l'API MHEG

Les Recommandations UIT-T de la série T.170 (et les parties de l'ISO/CEI 13522) partagent un nombre d'attributs clés. Les éléments MHEG suivants doivent être présents dans la spécification MHEG source:

– types de données MHEG, décrites en ASN.1 ou en Extended Backus-Naur Form (EBNF, *extended Backus-Naur form*);
– entités MHEG (à savoir les objets adressés par les actions MHEG élémentaires), liés les uns aux autres par des relations d'héritage;
– attributs statiques et dynamiques des entités MHEG;
– actions MHEG élémentaires s'appliquant aux entités MHEG;
– exceptions MHEG soulevées en tant qu'effet MHEG des actions élémentaires.

E.1.2 Eléments IDL en sortie du processus de définition de l'API MHEG

Le processus de définition de l'API va consister en un mappage entre ces éléments et un ensemble d'éléments IDL:

– aux types de données MHEG correspondront les types IDL non-objet;
– aux entités MHEG correspondront les interfaces d'objets IDL, reliés les uns aux autres par des relations d'héritage;
– aux attributs statiques et dynamiques des entités MHEG correspondront les attributs IDL, fournis par les interfaces d'objets IDL;
– aux actions MHEG élémentaires correspondront les opérations IDL, fournies par les interfaces d'objets IDL;
– aux exceptions MHEG produites sur effets d'actions élémentaires correspondront les exceptions IDL.

E.1.3 Contraintes vis-à-vis du processus de définition de l'API MHEG

Selon les directives ISO/CEI JTC 1 relatives à la normalisation des API, l'API MHEG sera définie comme une spécification abstraite d'API, à savoir une description, indépendante du langage, de la sémantique d'un ensemble de fonctionnalités en une syntaxe abstraite utilisant des types de données abstraits.

En accord avec les contraintes exprimées dans les recommandations du rapport technique ETR 225 "*API and script representation for MHEG – Requirements and framework*", une API MHEG satisfera aux contraintes suivantes:

– portabilité (voir E.1.3.1);
– généricité (voir E.1.3.2);
– caractère testable de la conformance (voir E.1.3.3);
– caractère implémentable de la représentation (voir E.1.3.4).

E.1.3.1 Portabilité

La contrainte de **portabilité** stipule que les applications MHEG ont besoin d'utiliser les services de manipulation et d'échange d'objets MHEG fournis par les moteurs MHEG (à savoir une API MHEG) d'une manière indépendante:

– du langage de programmation utilisé pour l'API MHEG;
– du système d'exploitation sous-jacent.

Pour répondre à cette contrainte, une API MHEG sera définie sous la forme d'une spécification abstraite d'API.

E.1.3.2 Généricité

La contrainte de **généricité** stipule que tous les besoins communs des applications MHEG doivent être supportés par une API MHEG.

Pour répondre à cette contrainte, une API MHEG sera définie au niveau le plus primitif, à savoir sous la forme de primitives qui correspondent aux actions MHEG élémentaires et de types de données qui correspondent aux types de données MHEG. Ceci garantit de maximiser le nombre des manipulations MHEG disponibles auprès des applications.

E.1.3.3 Caractère testable de la performance

Cette contrainte stipule qu'il devrait être aussi simple que possible de tester:

– la conformité d'un moteur MHEG à une spécification d'API MHEG, à savoir la fourniture correcte de cette API par un moteur MHEG en cours de test;

– la conformité d'une application MHEG à une spécification de l'API MHEG, à savoir l'utilisation correcte de cette API par une application MHEG en cours de test.

Pour répondre à cette contrainte, une API MHEG exprimera de manière formelle ses contraintes sur les implémentations et applications conformes, et utilisera une technique de description formelle pour la définition de l'API MHEG.

E.1.3.4 Caractère implémentable de la représentation

Cette contrainte stipule que les implémentations de moteurs MHEG dites conformes à la spécification de l'API MHEG doivent être aussi simples que possible. Ainsi, l'élaboration de l'API MHEG devrait prendre en compte des critères de simplicité et de clarté à la fois dans sa définition et sa formulation.

Pour répondre à cette contrainte, une API MHEG fournira ou référencera des directives de production de spécifications de mappage de langage et des règles de codage de message à partir de la spécification abstraite d'API.

E.1.3.5 Satisfaction des contraintes techniques

L'utilisation d'IDL contribue à satisfaire les contraintes techniques liées à la portabilité et au caractère testable:

– IDL est indépendante d'un langage de programmation. De plus, il existe des spécifications publiques de mappages IDL envers des langages de programmation courants comme C et C++;

– IDL fournit un langage de description formel complet autorisant une spécification d'API MHEG très concise, lisible et efficace. De plus, IDL est aussi appropriée pour la compilation automatique, de sorte que les implémentations d'API MHEG peuvent être automatiquement générées, pour un langage et un compilateur donnés, en utilisant les compilateurs adéquats.

E.1.4 Structure générale de l'API MHEG

L'API MHEG sera définie à l'aide d'IDL conformément à l'ISO/CEI 14750-1 [8]. L'utilisation d'IDL n'implique pas un environnement CORBA (*common object request broker architecture*). Ainsi, à côté de l'utilisation d'IDL, l'API MHEG sera définie indépendamment de l'architecture CORBA.

La définition d'une interface IDL sera composée de deux modules fournissant des fonctionnalités d'API différentes. Le premier module fournira les services d'interprétation MHEG. Ce module s'appellera **MHEG_<part>** où **<part>** est un nombre désignant la Recommandation adressée par l'API. Le second module fournira les services "accessor" et "modifier" pour les entités MHEG codées. Ce

module s'appellera **MHEG_<part>_Access** où **<part>** est un nombre désignant la Recommandation adressée par l'API.

L'API fournira une interface rendant possible la création d'une instance d'objet d'interface. Cette interface s'appellera **MHEGEntityManager**. Les opérations permettant la destruction d'une instance IDL seront fournies par les interfaces se rapportant aux entités MHEG.

Chaque interface se rapportant à une entité MHEG et fournissant un schéma de référence commun pour ses sous-classes fournira aussi les opérations suivantes:

– **attach**;

– **detach**;

– **getIdentifier**.

L'opération **attach** lie une instance d'objet d'interface avec une entité MHEG et retourne l'identificateur de l'objet lié. L'opération **detach** annule le lien. L'opération **getIdentifier** récupère l'identificateur MHEG de l'entité liée avec l'instance d'objet d'interface.

Les concepts IDL sont utilisés pour réaliser des interfaces de mappage avec les objets MHEG conformément aux règles décrites dans les sous-paragraphes qui suivent. Les identificateurs IDL assureront le mappage avec les identificateurs ASN.1 de MHEG conformément aux règles décrites dans les sous-paragraphes qui suivent.

Les opérations suivantes seront synchrones: create, bind, unbind, accessors et modifiers, GET. Aux autres actions élémentaires correspondront des opérations asynchrones.

La définition de l'API n'utilisera pas de niveaux de portée encapsulés.

E.1.5 Définition de types de données IDL non objets

Aux types de données MHEG exprimés en ASN.1 correspondront des types IDL non objet.

E.1.5.1 Mappage de nom

E.1.5.1.1 Types de données

Aux noms des types de données ASN.1 correspondront des noms de types IDL comme suit:

– si le nom du type ASN.1 est composé de plus d'un mot, le nom IDL ne contiendra pas de séparateur,

– chaque mot commencera avec une majuscule, toutes les autres lettres seront des minuscules.

Exemple: au nom ASN.1 **type-name** correspondra le nom IDL **TypeName**.

E.1.5.1.2 Composants

Aux composants ASN.1 correspondront des noms de champs IDL comme suit:

– si le nom ASN.1 est formé de un ou plusieurs mots, le caractère souligné ("_") sera utilisé comme séparateur;

– toutes les lettres seront des minuscules;

– comme IDL n'est pas "case-sensitive", des collisions peuvent se produire entre des noms de champs et de types lorsque des mots uniques sont utilisés. Dans un tel cas, le nom de champ sera préfixé par "the_". La même règle sera utilisée lorsqu'un nom de composant à mot unique rentre en collision avec un mot-clé IDL.

Exemple:

// regular field name

```
TypeName    field_name

// collision between field name and type name
Alias       the_alias

// collision between field name and IDL keyword ("string")
Type        the_string
```

E.1.5.1.3 Valeurs

Aux noms des valeurs ASN.1 correspondront des noms de valeurs IDL comme suit:

a) si le nom ASN.1 est formé de un ou plusieurs mots, le caractère souligné ("_") sera utilisé comme séparateur;

b) toutes les lettres seront des majuscules;

c) comme une énumération IDL ne crée pas de nouveau domaine, les valeurs utilisées à l'intérieur des énumérations peuvent provoquer des collisions. Dans un tel cas, tous les noms de valeurs d'une des énumérations rentrant en collision, et pas seulement le nom de la valeur pour laquelle se produit la collision, auront pour préfixe le nom du type énuméré. Ce nom de type sera créé selon les règles a) et b), et non selon les règles normales de nom de types.

Exemple:

```
// regular value name
FIRST_VALUE

// value name in case of a collision
FIRST_ENUMERATION_FIRST_VALUE
```

E.1.5.2 Mappage de type

E.1.5.2.1 INTEGER (entier)

A un type ASN.1 **INTEGER** correspondra un type IDL **long**.

Exemple:

```
-- ASN.1
Type ::= INTEGER

// IDL
typedef long Type;
```

E.1.5.2.2 BOOLEAN (booléen)

A un type ASN.1 **BOOLEAN** correspondra un type IDL **boolean**.

Exemple:

```
-- ASN.1
Type ::= BOOLEAN

// IDL
typedef boolean Type;
```

E.1.5.2.3 OCTET STRING (chaîne d'octets)

A un type ASN.1 **OCTET STRING** correspondra un type IDL **sequence <octet>**.

Exemple:

```
-- ASN.1
Type ::= OCTET STRING

// IDL
typedef sequence <octet> Type;
```

E.1.5.2.4 ENUMERATED (énumération)

A un type ASN.1 **ENUMERATED** correspondra un type IDL **enum**.

Exemple:

```
-- ASN.1
Type ::= ENUMERATED {value_1 (1), value_2 (2)}

// IDL
enum Type (VALUE_1, VALUE_2);
```

E.1.5.2.5 SEQUENCE OF (séquence de)

A un type ASN.1 **SEQUENCE OF** correspondra un type IDL **sequence**.

Exemple:

```
-- ASN.1
Type ::= SEQUENCE OF OtherType

// IDL
typedef sequence <OtherType> Type;
```

E.1.5.2.6 CHOICE (choix)

A un type ASN.1 **CHOICE** correspondra une union IDL discriminante obtenue par combinaison des types **enum** et **union**. Le nom de type **enum** sera dérivé du nom du type **CHOICE** suffixé par “Tag” afin de ne pas rentrer en collision avec le nom de type **union**. Les noms de valeurs **enum** seront dérivés des noms, des champs de type choix ayant le suffice "_TAG" de manière à éviter toute collision avec les noms de champs à l’intérieur du **switch**. Dans un **CHOICE**, à un composant **NULL** correspondra un **case** vide.

Exemple:

```
-- ASN.1
Type ::= CHOICE
{
        a       A,
        b       B,
        c       NULL
}

// IDL
enum TypeTag { A_TAG, B_TAG, C_TAG };
union Type
switch TypeTag {
        case A_TAG:
                A       the_a;
        case B_TAG:
                B       the_b;
        // no’case’ for C_TAG
};
```

E.1.5.2.7 SEQUENCE (séquence)

A un type **SEQUENCE** ASN.1 correspondra une type **struct** IDL. Aux composants **OPTIONAL** ASN.1 avec ou sans valeurs **DEFAULT** correspondra une **sequence** IDL d'au moins un élément de ce type.

Exemple:

```
-- ASN.1
Type ::= SEQUENCE
{
        a       A OPTIONAL,
        b       B,
        c       INTEGER DEFAULT 0
}

// IDL
struct Type {
        sequence <A,1>          the_a;
        B                       the_b;
        sequence <long,1>       c;
};
```

E.1.5.3 Ordre des déclarations

L'ordre des déclarations de types peut être différent entre ASN.1 et IDL car IDL ne permet pas d'utiliser un type avant qu'il ne soit déclaré (ne fournit pas la fonctionnalité de référence en avant). Les déclarations de types IDL seront réorganisées pour faire face à cette contrainte.

Afin de permettre cette réorganisation, les références croisées à partir de la syntaxe ASN.1 seront supprimées.

Les exemples suivants montrent des références croisées couramment utilisées en ASN.1 et montrent aussi comment elles peuvent être supprimées.

Exemple 1:

```
-- ASN.1 definition using a cross-reference
-- Production rules used :
-- T = A | B
-- B = T C
-- Allowed values :
-- a c*

-- T refers to B
T ::= CHOICE
{
        a       A,
        b       B
}
-- B refers to T
B ::= SEQUENCE
{
        t       T,
        c       C
}

-- Equivalent ASN.1 definition without cross-reference
-- The information described remains the same, the structure has changed
-- Production rules used :
-- T = A C*
```

```
T ::= SEQUENCE
{
        a       A,
        s_o_c   SEQUENCE OF C OPTIONAL
}
```

Exemple 2:

```
-- ASN.1 definition using a cross-reference
-- Production rules used :
-- T = A ⏐ B
-- B = C T
-- Allowed values :
-- c*a

-- T refers to B
T ::= CHOICE
{
        a       A,
        b       B
}
-- B refers to T
B ::= SEQUENCE
{
        c       C,
        t       T
}

-- Equivalent ASN.1 definition without cross-reference
-- The information described remains the same, the structure has changed
-- Production rules used :
-- T = C* A

T ::= SEQUENCE
{
        s_o_c   SEQUENCE OF C OPTIONAL,
        a       A
}
```

Cependant, dans la plupart des cas, les références croisées devraient être enlevées en créant des types encapsulés, comme le montre l'exemple 3.

Exemple 3:

```
-- ASN.1 definition using a cross-reference

-- T refers to A
T ::= CHOICE
{
        a       A,
        b       B
}

A ::= -- A definition that refers to T (either directly or indirectly)

-- Equivalent ASN.1 definition without cross-reference
```

```
T ::= CHOICE
{
        a       -- A definition
        b       B
}
```

Le désavantage de la méthode montrée dans l'exemple 3 réside dans le fait que des types de données complexes peuvent être créés. De plus, une partie des types de données ainsi créés peut dupliquer des types déjà définis dans la syntaxe.

Une autre option est de traduire le type de donnée ASN.1 "A" dans une interface IDL "A" encapsulant le type de donnée IDL "A". Ceci permet d'utiliser une déclaration en avant de l'interface IDL "A" comme l'illustre l'exemple 4.

Exemple 4:

```
interface A; // forward declaration

enum TTag { A_TAG, B_TAG };
union T
switch TTag {
        case A_TAG:
                A       the_a;
        case B_TAG:
                B       the_b;
};

interface A {

// this interface embeds the definition of type A which refers to type T

};
```

E.1.6 Définition d'interface IDL

Aux entités MHEG correspondront des interfaces IDL conformément aux règles suivantes:

– chaque élément pouvant être désigné comme destination d'une action MHEG élémentaire sera considéré comme une entité. Les entités ne sont pas toutes explicitement définies par la norme MHEG. De plus, elles n'existent pas toutes dans le modèle d'objet MHEG. Par conséquent, le modèle d'objet MHEG sera complété pour intégrer toutes les entités utilisées;

– lorsqu'une action élémentaire pourra s'adresser à différents types d'entités, une nouvelle entité sera créée en agrégeant ces entités;

– le nom de l'entité nouvellement créée sera formé par la concaténation des noms des entités agrégées en utilisant "Or" comme séparateur;

– le modèle d'objet obtenu est le modèle d'objet de l'API. Il y aura une correspondance un pour un entre les entités et les interfaces;

– l'arbre d'héritage IDL correspondra à la hiérarchie du modèle d'objet de l'API;

– les noms d'interfaces seront dérivés des noms d'entités en utilisant les mêmes règles que pour les types de données.

E.1.7 Définition d'attribut IDL

Les catégories suivantes d'attributs MHEG ont été identifiées:

– attributs échangés;

– attributs internes.

E.1.7.1 Attributs MHEG échangés

Aux attributs MHEG échangés correspondront des attributs IDL dans le module **MHEG-<part>-access** selon les règles suivantes:

a) à chaque type ASN.1 dont le nom se termine par "Class" correspondra une interface;

b) à l'intérieur du type [voir a)], aux **COMPONENTS OF** ASN.1 correspondra la fonctionnalité d'héritage IDL;

c) à l'intérieur du type [voir a)], à chaque composant correspondra un attribut;

d) le nommage et le typage des attributs suivront les règles définies au E.1.5.

Exemple:

```
-- ASN.1

B-Class ::= SEQUENCE
{
        COMPONENTS OF       A-Class,
        i                   INTEGER,
        j                   C-Type
}

// IDL

interface BClass : AClass {
        attribute long      i;
        attribute CType     j;
};
```

E.1.7.2 Attributs MHEG internes

On ne fera pas correspondre aux attributs IDL des attributs MHEG internes. Les attributs MHEG internes sont accédés et modifiés par le biais d'actions élémentaires MHEG auxquelles correspondront des opérations IDL.

E.1.8 Définition d'opération IDL

Les interfaces de l'API MHEG fourniront les catégories d'opérations suivantes:

– opérations faisant correspondre les actions élémentaires MHEG à destination de l'entité MHEG à laquelle correspond l'interface;

– opérations pour détruire une instance d'interface (voir Note);

– opérations utilisées pour attacher ou détacher une instance d'interface d'une entité MHEG.

NOTE – La création d'une instance d'interface est fournie par l'interface de fabrication.

E.1.8.1 Opérations faisant correspondre des actions élémentaires MHEG

Aux actions MHEG élémentaires correspondront des opérations IDL conformément aux règles suivantes:

a) il y aura une correspondance un-à-un entre les actions élémentaires et les opérations;

b) l'interface fournie par une opération correspondra à l'entité destination de l'action élémentaire;

c) le nom de l'opération sera décliné du nom de l'action élémentaire selon le schéma suivant:

   1) si le nom de l'opération est formé de plus d'un mot, on n'utilisera pas de séparateur;

   2) le nom de l'opération commencera par une lettre minuscule;

3) chaque mot (sauf le premier) commencera par une lettre majuscule.

d) à une action élémentaire "Set" correspondra une opération selon le schéma suivant:

1) la valeur de retour de l'opération sera de type **void**;

2) il y aura seulement des paramètres d'entrée;

3) les types de données et les noms des paramètres seront dérivés de la définition de l'action élémentaire ASN.1 en utilisant les règles définies au E.1.5;

4) on n'établira pas de correspondance pour le premier paramètre de la définition de l'action élémentaire ASN.1 (il représente la destination de l'action élémentaire);

5) on utilisera le type de donnée encapsulé lorsque la fonctionnalité est utilisée pour la définition d'un paramètre;

6) si le type de donnée référencée dans d 5) n'est pas explicitement défini, il sera créé comme dans l'exemple 1 au E.1.5.3.

e) à une action élémentaire "Get" correspondra une opération selon le schéma suivant:

1) la valeur de retour de l'opérateur sera celle évaluée par l'action élémentaire;

2) le type de la valeur de retour n'existe pas en tant que tel dans la syntaxe ASN.1 car les actions élémentaires évaluent des valeurs génériques. Par conséquent, il sera l'un parmi les suivants:

i) un type de donnée simple;

ii) un type de donnée complexe utilisé dans l'action élémentaire "Set" correspondante;

iii) si un type de donnée complexe est retourné sans qu'il existe d'action élémentaire "Set" correspondante, le type de donnée sera créé;

3) il n'y aura pas de paramètre de sortie, tous les résultats seront passés en utilisant la valeur de retour;

4) pour les paramètres d'entrée, les règles définies dans d 3), d 4), d 5) et d 6) s'appliquent.

Exemple:

```
Elementary-Action-N ::=  SEQUENCE
{
        target          Target, -- to be skipped
        param1-param    Param1-Parameter, -- replace
        param2-param    Param2-Parameter -- replace
}

Param1-Parameter ::= CHOICE
{
        param1          Param1,
        param1-macro    Param1-Macro
}

Param1-Macro ::=  SEQUENCE
{
        -- ...
        -- ...
}
Param1 ::=   -- ...

--      In "Elementary-Action-N" the "param1-parameter" parameter shall be replaced with the
--      "param1" parameter
```

```
Param2-Parameter ::= CHOICE
{
        a-param                 A-Parameter,
        b-param                 B-Parameter
}

A-Parameter ::= CHOICE
{
        a                       A,
        a-macro                 A-Macro
}

A-Macro ::= SEQUENCE
{
        -- ...
        -- ...
}

A ::=           -- ...

B-Parameter ::=   SEQUENCE
{
        -- ...
        -- ...
}

B-Macro ::= SEQUENCE
{
        -- ...
        -- ...
}

B ::=           -- ...

--      In "Elementary-Action-N" the "param2-parameter" parameter cannot be replaced with the
--      "param2" parameter because it does not exist. Consequently an IDL datatype that
--      corresponds to the following ASN.1 datatype shall be created:

Param2 ::= CHOICE
{
        a               A,
        b               B
}
```

E.1.8.2 Opérations rendant possible la destruction d'une instance d'interface

Chaque interface fournira une opération rendant possible la destruction d'une instance d'interface. Cette opération aura le prototype suivant:

```
void    kill();
```

E.1.8.3 Opérations permettant d'attacher et détacher une instance d'interface d'une entité MHEG

Chaque interface fournira des opérations rendant possible d'attacher et de détacher une instance d'interface à une entité MHEG. Ces opérations seront nommées respectivement **attach** et **detach**. L'opération **attach** acceptera une référence à une entité MHEG comme paramètre, résoudra cette référence et retournera un identificateur d'entité MHEG. L'opération **detach** n'acceptera pas de paramètre.

Chaque interface fournira une opération pour récupérer l'identificateur de l'entité MHEG attaché à une instance d'interface. Cette opération sera nommée **getIdentifier**. Elle n'acceptera aucun paramètre et retournera un identificateur d'entité MHEG.

E.1.9 Définition d'exception IDL

Aux exceptions IDL correspondront des conditions d'erreur MHEG associées aux actions MHEG élémentaires. Les conditions d'erreur devraient être classées selon leurs natures de manière à en obtenir un nombre limité.

L'ensemble d'exceptions suivant est recommandé:

– **InvalidTarget**;

– **InvalidParameter**;

– **NotBound**;

– **AlreadyBound.**

L'exception **InvalidTarget** apparaît lorsque l'entité MHEG destination n'est pas disponible. Le membre **period** retourne l'état courant (période de cycle de vie) de la destination.

L'exception **InvalidParameter** apparait lorsque la valeur d'un des paramètres empêche l'exécution normale de l'action. Le membre **completion_status** indique si l'action s'est terminée (avec une valeur par défaut affectée au paramètre non conforme) ou non. Le membre **parameter_number** indique le rang du paramètre non valide.

L'exception **NotBound** apparait lorsque l'instance de l'objet d'interface n'est liée à aucune entité MHEG.

L'exception **AlreadyBound** apparait lorsque l'instance de l'objet d'interface est déjà liée avec une entité MHEG. Le membre **entity_identifier** identifie l'entité liée.

La définition de ces exceptions s'effectue comme suit:

```
exception InvalidTarget {
        unsigned short period;
};

enum CompletionStatus { YES, NO };

exception InvalidParameter {
        CompletionStatus   completion_status;
        unsigned short     parameter_number;
};

typedef long EntityIdentifier;

exception AlreadyBound {
        EntityIdentifier entity_identifier;
};

exception NotBound {};
```

Les exceptions mentionnées ci-dessus seront définies avec une portée globale.

Si une condition d'erreur ne peut pas être prise en compte par l'une des exceptions mentionnées ci-dessus, une exception spécifique sera créée au niveau de portée d'interface approprié.

Les noms d'exceptions IDL des exceptions spécifiques seront dérivés du nom de la condition d'erreur MHEG en utilisant les même règles que celles des types de données:

– pas d'utilisation de séparateur;

– le nom d'exception commencera par une lettre majuscule;

– s'il est formé de plus d'un mot, il n'y aura pas de séparateur entre les mots et chaque mot commencera par une lettre majuscule.

E.2 Mappage entre l'API MHEG et la SIR MHEG

Produire une API MHEG correspondant à la SIR MHEG est un processus consistant à affecter des identificateurs de types prédéfinis, des identificateurs de fonctions prédéfinies et des identificateurs de messages prédéfinis pour mapper aux types, opérations et exceptions de l'API MHEG.

Ce processus de mappage sera composé de trois étapes successives:

a) extraire les types intermédiaires jusqu'à ce que toutes les définitions de types aient un seul niveau (voir exemple 1 au E.1.5.3);

b) changer les types IDL **enum** en **unsigned long** (la SIR MHEG ne fournit aucun type de donnée énuméré). Calculer son numéro de "niveau" pour chaque définition de type, à savoir la distance de la feuille la plus grande possible. Unifier les types équivalents, en commençant par ceux de numéro de "niveau" le plus bas;

c) scruter la spécification IDL résultante pour allouer les ID prédéfinis. Affecter les TIDs et MIDs prédéfinis dès la rencontre de nouveaux types ou de nouvelles exceptions. Pour chaque objet, affecter des FIDs prédéfinis à chaque opération explicitement définie, à chaque opération héritée n'étant pas remplacée par une autre opération définie, à chaque accesseur d'attribut et à chaque modificateur d'attribut.

Exemple:

```
struct A {
        int a;
        struct {
                int b;
                struct {
                        int c;
                        int d;
                } b
        } c
};

// after extraction of intermediate types (a))

struct A {
        int a;
        B b;
};
struct B {
        int b;
        C c;
};
struct C {
        int c;
        int d;
};
```

ANNEXE F

Spécification IDL de l'API MHEG-3

Pour exprimer la manipulation directe d'autres scripts, les scripts échangés utiliseront le mappage à MHEG-3 (tel que défini dans l'Annexe B) de l'API MHEG-3.

Les moteurs MHEG-3 fourniront l'API MHEG-3 telle qu'elle est spécifiée par le module IDL **MHEG_3**.

```
module MHEG_3 {

    enum RtScriptStatus {RUNNING, READY, ERRONEOUS};
    enum InvocationStatus {NOT_STARTED, PROCESSING, TERMINATED, ABORTED};
    enum PassingMode {BY_VALUE, BY_REFERENCE};
    enum Entity {TYPE, DATA, FUNCTION, MESSAGE, PACKAGE, HANDLER};

    typedef unsigned short FID;
    typedef unsigned short DID;
    typedef unsigned short TID;

    struct ContentReference {
        string public_id;
        string system_id;
    };
    struct ParameterDescription {
        PassingMode     passing_mode;
        TID             parameter_type_id;
    };
    struct Prototype {
        TID             return_value_type_id;
        sequence<ParameterDescription> signature;
    };

    exception InvalidParameter {
        unsigned short rank;
    };
    exception InvalidScript {
        Entity      the_entity;
        unsigned short identifier;
    };
    exception OperationFailed {};

    interface MhScript;
    interface RtScript;
    interface RoutineInvocation;

    interface ScriptInterpreter {
        MhScript prepare (in ContentReference content_reference)
            raises (InvalidScript, InvalidParameter, OperationFailed);
        void kill ();
    };

    interface MhScript {
        RtScript new ()
            raises (OperationFailed);
        void destroy ();
    };
```

```
        interface RtScript {
                RoutineInvocation open (in FID routine_id)
                        raises (InvalidParameter);

                RtScriptStatus getRtScriptStatus();

                void setPriority (in unsigned short priority);
                unsigned short getPriority ();

                DID allocate (in TID variable_type_id)
                        raises (InvalidParameter, OperationFailed);
                void free (in DID variable_id)
                        raises (InvalidParameter);

                void setData (in DID variable_id, in any variable_value)
                        raises (InvalidParameter, OperationFailed);
                any getData (in DID data_id)
                        raises (InvalidParameter, OperationFailed);

                void stop ()
                        raises (OperationFailed);
                void reInit ()
                        raises (OperationFailed);
                void delete ();
        };

        interface RoutineInvocation     {

                readonly attribute FID routine_id;

                void setParameter (in unsigned short rank, in TID parameter_type_id,
                                        in any parameter_value)
                        raises(InvalidParameter);

                Prototype getPrototype ();
                InvocationStatus getInvocationStatus ();

                void run ()
                        raises (OperationFailed);

                void reset ();
                void close ();
        };
};
```

ANNEXE G

Relations avec les autres parties des Recommandations UIT-T de la série T.170 (et parties de l'ISO/CEI 13522)

G.1 Relations avec la Rec. UIT-T T.171 (et l'ISO/CEI 13522-1)

Le présent sous-paragraphe spécifie les conditions qu'il est requis d'appliquer lorsque la présente Recommandation est utilisée pour étendre les dispositions de la Rec. UIT-T T.171 (et l'ISO/CEI 13522-1) [5].

L'affirmation "une autre Recommandation UIT-T de la série T.170" ("une des parties de l'ISO/CEI 13522") utilisée dans la présente Recommandation sera interprétée comme étant "Rec. UIT-T T.171" ("ISO/CEI 13522-1").

La SIR MHEG définie par la présente Recommandation s'appliquera au codage du composant **script-data** des objets de la classe **script** conformes à la Rec. UIT-T T.171 (ISO/CEI 13522-1), dont le composant **script-classification** est enregistré dans le catalogue **registered-script-classification** à l'indication "script" et dont le composant **catalogued-script-encoding** est enregistré dans le catalogue **registered-script-classification** à l'indication "SIR MHEG", conformément aux valeurs maintenues par l'autorité d'enregistrement MHEG.

NOTE – Ces valeurs sont enregistrées conformément aux dispositions de l'ISO/CEI 13522-4 [6].

Dans ce cadre, le composant **script-data** sera codé conformément aux restrictions suivantes sur le module **ISOMHEG-sc** défini dans la Rec. UIT-T T.171 (ISO/CEI 13522-1) [5]:

– si le choix **script-inclusion** a été sélectionné pour le composant **script-data**, alors le choix **interchangedscript** sera sélectionné pour le composant **script-inclusion**;

– si le choix **data-reference** a été sélectionné pour le composant **script-data** alors les données référencées seront codées comme un **InterchangedScript** conformément à la spécification du module **ISO-MHEG-sir**.

Ainsi les objets script MHEG-3 seront conformes à la syntaxe définie par le sous type **SIR-Script-Class** ci-après.

```
-- ISOMHEG-sir {joint-iso-itu-t(2) mheg (19) version (1) script-interchanged-representation (11)}

IMPORTS
        MHEG-Identifier, Catalogued-Script-Classification
                FROM ISOMHEG-ud { joint-iso-itu(2) mheg(19) version(1) useful-definitions (9) }
        Registered-Script-Encoding, Registered-Script-Classification
                FROM ISOMHEG-cat { joint-iso-itu(2) mheg(19) version(1) catalogues (12) }
        Script-Class
                FROM ISOMHEG-sc { joint-iso-itu(2) mheg(19) version(1) script-class(3) }
;

SIR-Script-Class ::= Script-Class (WITH COMPONENTS
{       ...,
        script-classification (WITH COMPONENTS
        {       registered-script-classification    ('script') }) PRESENT,
                                        -- registered value as provided by ISOMHEG-cat

        script-hook (WITH COMPONENTS
        {       ...,
                catalogued-script-encoding (WITH COMPONENTS
                {       registered-script-encoding ('MHEG-SIR') }) PRESENT,
                                        -- registered value as provided by ISOMHEG-cat
        }) PRESENT,

        script-data (WITH COMPONENTS
        {       ...,
                script-inclusion (WITH COMPONENTS       { interchangedscript }),
                data-reference (WITH COMPONENTS         { null-data ABSENT })
        }) PRESENT
})
```

G.2 Relations avec la Rec. UIT-T T.172 (et l'ISO/CEI 13522-5)

Le présent sous-paragraphe spécifie les conditions qu'il est requis d'appliquer lorsque la présente Recommandation est utilisée pour étendre les dispositions de la Rec. UIT-T T.172 (et l'ISO/CEI 13522-5) [7].

L'affirmation "une autre Recommandation UIT-T de la série T.170" ("une des parties de l'ISO/CEI 13522") utilisée dans la présente Recommandation sera interprétée comme étant "Rec. UIT-T T.172" ("ISO/CEI 13522-5").

La SIR MHEG définie par la présente Recommandation s'appliquera au code de l'attribut **original-content** des objets **InterchangedProgram** conformes à la Rec. UIT-T T.172 (et l'ISO/CEI 13522-5) dont le composant **content-hook** mentionne "MHEG-SIR" selon la définition du domaine d'application.

Au sein de la valeur du type **InterchangedProgramClass**, les restrictions suivantes s'appliqueront au module **ISOMHEG-MHEG-5** défini par la Rec. UIT-T T.172 (et l'ISO/CEI 13522-5) [7]:

– dans le cas où **included-content** est sélectionné pour le composant **original-content**, alors la valeur **OCTET STRING** du composant **included-content** sera remplacée par une valeur du type **InterchangedScript** selon la spécification du module **ISO-MHEG-sir**;

– dans le cas où **referenced-content** est sélectionné pour le composant **original-content** alors la donnée référencée sera une valeur du type **InterchangedScript** selon la spécification du module **ISO-MHEG-sir**.

Donc, les objets scripts MHEG-3 seront conformes à la syntaxe définie par le sous type **SIR-Script-Class** ci-après.

```
ISOMHEG-sir { joint-iso-itu-t(2) mheg (19) version (1) script-interchanged-representation (11) }

IMPORTS
        InterchangedProgramClass
                FROM ISOMHEG-MHEG-5 { joint-iso-itu(2) mheg(19) version(1) MHEG-5 (17) }
;

SIR-Script-Class ::= InterchangedProgramClass (WITH COMPONENTS
{       ...,
        content-hook ('MHEG-SIR')  PRESENT,

        original-content PRESENT,
                -- data encoded as InterchangedScript
})
```

La Rec. UIT-T T.172 (et l'ISO/CEI 13522-5) [7] ne fait aucune distinction entre les "mh-objects" et les "rt-objects". Tous les deux sont appelés "objets MHEG-5". Il en résulte qu'il n'y aura qu'un "rt-script" par "mh-script". Par conséquent:

– les initialisations de "mh-script" et de "rt-script" seront effectuées dans une opération unique (correspondant à l'invocation des opérations **prepare** et **new**);

– le fait d'invoquer plusieurs opérations new sur le même "mh-script" fera apparaître une exception;

– les destructions de "rt-script" et de "mh-script" seront effectuées dans une opération unique (correspondant à l'invocation des opérations **delete** et **destroy**).

APPENDICE I

Syntaxe de la SIR MHEG (notation EBNF)

Le présent appendice décrit la syntaxe des scripts SIR MHEG échangés. Cette syntaxe est équivalente à la spécification ASN.1 de l'Annexe A.

// Structure

```
InterchangedScript      ::=  TypeDeclaration*
                             ConstantDeclaration*
                             VariableDeclaration*
                             PackageDeclaration*
                             HandlerDeclaration*
                             RoutineDeclaration*

// Type declarations
TypeDeclaration         ::=  TypeIdentifier?
                             TypeDescription

TypeDescription         ::=  SequenceDescription
                        |    StringDescription
                        |    ArrayDescription
                        |    StructureDescription
                        |    UnionDescription

SequenceDescription          ::=  INTEGER?          // Sequence bound
                             TypeIdentifier

StringDescription       ::=  INTEGER?          // String bound

ArrayDescription        ::=  INTEGER           // Array size
                             TypeIdentifier

UnionDescription        ::=  TypeIdentifier+

StructureDescription         ::=  TypeIdentifier+

// Data declarations
ConstantDeclaration     ::=  DataIdentifier?
                             TypeIdentifier
                             ConstantValue

ConstantValue           ::=  BOOLEAN
                        |    OCTET
                        |    INTEGER           // all numeric types
                        |    REAL              // float or double
                        |    STRING            // character or string
                        |    DataIdentifier
                        |    ConstantValue*          // sequence, array or structure
                        |    UnionValue

UnionValue              ::=  INTEGER           // Tag index
                             ConstantValue

VariableDeclaration          ::=  DataIdentifier?
                             TypeIdentifier
                             ConstantReference?     // Initial value

ConstantReference       ::=  DataIdentifier
                        |    ConstantValue

// Package declarations
PackageDeclaration      ::=  PackageIdentifier?
                             VisibleString          // Package name
                             ServiceDescription*
                             ExceptionDescription*

ServiceDescription      ::=  FunctionIdentifier?
                             VisibleString?         // IDL global name
```

```
                    CallingMode?
                    TypeIdentifier?     // return value
                    ParameterDescription*

ServiceParameterDescription     ::=     ServicePassingMode?
                    TypeIdentifier

CallingMode         ::=     'SYNCHRONOUS' | 'ASYNCHRONOUS'
ServicePassingMode      ::=     'IN' | 'OUT' | 'INOUT'

ExceptionDescription            ::=     MessageIdentifier?
                    VisibleString?          //IDL exception global name
                    TypeIdentifier*     //Parameter types

// Handler declarations
HandlerDeclaration      ::=     MessageIdentifier
                    FunctionIdentifier

// Routine declarations
RoutineDeclaration      ::=     FunctionIdentifier?
                    TypeIdentifier?     // for return value
                    RoutineParameterDescription*
                    VariableDeclaration*
                    OCTET STRING        // program code

RoutineParameterDescription     ::=     RoutinePassingMode?
                        TypeIdentifier

RoutinePassingMode      ::=     'VALUE' | 'REFERENCE'

// Useful definitions
TypeIdentifier          ::=     INTEGER
DataIdentifier          ::=     INTEGER
FunctionIdentifier      ::=     INTEGER
MessageIdentifier       ::=     INTEGER
PackageIdentifier       ::=     INTEGER
```

APPENDICE II

Notation textuelle des scripts de la SIR MHEG

Le présent appendice décrit une notation textuelle lisible pour l'expression des scripts SIR MHEG. Elle peut être utile pour exprimer des exemples de SIR MHEG. Elle est donnée en notation EBNF.

```
InterchangedScript      ::=     "SCRIPT "
                    TypeDeclaration*
                    ConstantDeclaration*
                    VariableDeclaration*
                    PackageDeclaration*
                    HandlerDeclaration*
                    RoutineDeclaration*
                    "ENDSCRIPT "

// Type declarations
TypeDeclaration     ::=     "TYPE "
                    Identification?
                    TypeDescription
                    "ENDTYPE "
```

```
TypeDescription         ::=   "SEQUENCE "     SequenceDescription   "ENDSEQUENCE "
                        |     "STRING "  StringDescription   "ENDSTRING "
                        |     "ARRAY "   ArrayDescription    "ENDARRAY "
                        |     "STRUCT "  StructureDescription      "ENDSTRUCT "
                        |     "UNION "   UnionDescription    "ENDUNION "

SequenceDescription             ::=   INTEGER          // Sequence bound
                        Reference                 // Type identifier

StringDescription       ::=   INTEGER?         // String bound

ArrayDescription        ::=   INTEGER          // Array size
                        Reference                 // Type identifier

UnionDescription        ::=   Reference+       // Type identifiers

StructureDescription            ::=   Reference+       // Type identifiers

// Data declarations
ConstantDeclaration     ::=   "CONSTANT "
                        Identification?
                        Reference          // Type identifier
                        ConstantValue
                        "ENDCONSTANT "

ConstantValue           ::=   "BOOLEAN "      BOOLEAN
                        |     "OCTET "   OCTET
                        |     "SHORT "   INTEGER
                        |     "LONG "    INTEGER
                        |     "WORD "    INTEGER
                        |     "UNSIGNED "      INTEGER
                        |     "FLOAT "   REAL
                        |     "DOUBLE "REAL                              |     "CHAR "    STRING
                        |     "STRING "  STRING
                        |     "IDENTIFIER "    Reference    // Data identifier
                        |     "SEQUENCE "      ConstantValue*
                        |     "STRUCT " ConstantValue*
                        |     "ARRAY "   ConstantValue*
                        |     "UNION "   UnionValue

UnionValue              ::=   INTEGER          // Tag index
                        ConstantValue

VariableDeclaration     ::=   "VARIABLE "
                        Identification?
                        Reference          // Type identifier
                        ConstantReference?       // Initial value (constant)
                        "ENDVARIABLE "

ConstantReference       ::=   Reference        // Data identifier
                        |     ConstantValue

// Package declarations
PackageDeclaration      ::=   "PACKAGE "
                        IntegerIdentification?
                        STRING?                // Package name
                        ServiceDescription*
                        ExceptionDescription*
                        "ENDPACKAGE "
```

```
ServiceDescription          ::=  "SERVICE "
                                 IntegerIdentification?
                                 STRING?                  // IDL operation global name
                                 CallingMode?
                                 Reference?               // Return type identifier
                                 ServiceParameterDescription*
                                 "ENDSERVICE "

ServiceParameterDescription ::=  "PARAM "
                                 ServicePassingMode
                                 Reference                // Type identifier

CallingMode                 ::=  "SYNC " | "ASYNC "
ServicePassingMode          ::=  "IN " | "OUT " | "INOUT "

ExceptionDescription        ::=  "EXCEPTION "
                                 IntegerIdentification?
                                 STRING?                  //IDL exception global name
                                 Reference*               // Parameter type identifiers
                                 "ENDEXCEPTION "

// Handler declarations
HandlerDeclaration          ::=  "HANDLER "
                                 Reference                // Message identifier
                                 Reference                // Function identifier
                                 "ENDHANDLER "

// Routine declarations
RoutineDeclaration          ::=  "ROUTINE "
                                 Identification?
                                 Reference?               // Return type identifier
        RoutineParameterDescription*
                                 VariableDeclaration*
                                 Instruction+
                                 "ENDROUTINE "

RoutineParameterDescription ::=  "PARAM "
                                 RoutinePassingMode?
                                 Reference                // Type identifier

RoutinePassingMode          ::=  "VAL " | "VAR "

// Useful definitions
Identification              ::=  "ID " Reference
IntegerIdentification       ::=  "ID " INTEGER

INTEGER                     ::=  DecimalInteger | HexaInteger
DecimalInteger              ::=  Sign? Digit+ " "
Sign                        ::=  "+" | "-"
Digit                       ::=  "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
HexaInteger                 ::=  "h" Hex+
Hex                 ::=     Digit | "A" | "B" | "C" | "D" | "E" | "F"
BOOLEAN                     ::=  "TRUE " | "FALSE "
OCTET                       ::=  Hex Hex " "
STRING                      ::=  """ Character* """
Character                   ::=  ... // visible character
REAL                        ::=  Sign? Digit+ "." Digit* "e" Sign? Digit+  " "

// Program code
Instruction                 ::=  SimpleInstruction " "
                        |        LogicalOperator LogicalTypeCode " "
```

```
                |   "EQ_" TypeCode " "
                |   ComparisonOperator ComparisonTypeCode " "
                |   ArithmeticOperator NumericTypeCode  " "
                |   "NEG_" SignedTypeCode " "
                |   "REM_" IntegerTypeCode "´"
                |   "DUP_" TypeCode " "
                |   "CVT_" ConvertibleTypes " "
                |   "SHIFT_" UnsignedTypeCode "´" INTEGER
                |   JumpInstruction " " Destination
                |   UnaryInstruction " " Reference
                |   "PUSHI " INTEGER
                |   BinaryInstruction " " Reference INTEGER
                |   "LABEL " STRING        // Label in routine code

IntegerTypeCode         ::=   SignedIntegerTypeCode | UnsignedTypeCode
SignedIntegerTypeCode           ::=   "S" | "L"
SignedTypeCode          ::=   SignedIntegerTypeCode | "F" | "D"
UnsignedTypeCode        ::=   "O" | "W" | "U"
LogicalTypeCode         ::=   "B" | UnsignedTypeCode
NumericTypeCode         ::=   SignedTypeCode | UnsignedTypeCode
ComparisonTypeCode      ::=   NumericTypeCode | "C"
ConvertibleTypes        ::=   "SW" | "WS" | "LU" | "UL" | "CW" | "WC" | "BS" | "OS"
                |   "SL" | "LF" | "WL" | "UF" | "FD" | "BO" | "OW" | "SU"
                |   "WU" | "OB" | "SB" | "LB" | "WB" | "UB" | "WO" | "LS"
                |   "FL" | "UW" | "FU" | "DF"
TypeCode                ::=   ComparisonTypeCode | "B" | "I" | "R"

Reference               ::=   INTEGER          // Identifier
                |   STRING             // Logical name or IDL name
Destination             ::=   INTEGER          // Offset
                |   STRING             // Label

SimpleInstruction               ::=   "EQR" | "YIELD" | "RET" | "NOP" | "FREE"

ComparisonOperator              ::=   "GT_" |      "LT_"

LogicalOperator         ::=   "AND_" | "OR_" | "XOR_" | "NOT_"

ArithmeticOperator              ::=   "ADD_" | "SUB_" | "MUL_" | "DIV_"

JumpInstruction         ::=   "JMP" | "LJMP" | "JT" | "JF" | "LJT" | "LJF"

UnaryInstruction                ::=   "INC" | "DEC" | "PUSHR" | "PUSH" | "POPR" | "POP" |"POPC"
                |   "GETOR" | "ALLOC" | "CALL" | "XCALL"

BinaryInstruction               ::=   "SET" | "SETC" | "GET" | "GETC"
```

APPENDICE III

Entités MHEG

Les entités MHEG comprennent des objets MHEG, des "mh-objects", des "rt-objects", des objets MHEG échangés, des "sockets" et des "channels".

III.1 Objets MHEG

Selon la Rec. UIT-T T.172 (et l'ISO/CEI 13522-1) [5], un objet MHEG est défini comme une représentation codée. Par conséquent, les objets MHEG sont des trains binaires. L'identité d'un objet

MHEG est son train binaire. Les objets MHEG sont des objets de "form 1" selon la description qui en est faite au 6.2.4 de l'ISO/CEI 13522-1 [5]. Les objets MHEG A et B sont identiques si et seulement s'ils possèdent le même train binaire.

Un objet MHEG n'est pas un objet physique, mais une abstraction (une séquence spécifiée de bits) pouvant avoir plusieurs représentations (à savoir différents objets) de types différents: objets MHEG échangés, objets MHEG stockés, "mh-objects", etc. De telles représentations sont manipulées par différents services logiciels.

Un objet MHEG peut être identifié par un identificateur MHEG. Les identificateurs MHEG sont les **seuls** moyens d'identifier les objets MHEG. La structure et la représentation codée des identificateurs MHEG sont définies dans la Rec. UIT-T T.172 (et l'ISO/CEI 13522-1) [5]. L'identificateur d'un objet MHEG doit être codé à l'intérieur de l'objet MHEG. Comme l'attribut est optionnel, certains objets MHEG n'ont pas d'identificateur MHEG. De tels objets ne peuvent pas être identifiés. La Rec. UIT-T T.172 (et l'ISO/CEI 13522-1) [5] impose la contrainte suivante dans la réalisation des applications MHEG: les objets MHEG A et B ne devront pas avoir le même identificateur sauf s'ils sont identiques.

La référence MHEG générique décrit tous les moyens possibles de référencer un objet MHEG.

III.2 Mh-objects

Un "mh-object" est la représentation interne d'un objet MHEG à l'intérieur d'un processus ou d'un système. Un "mh-object" n'est pas un objet MHEG. A l'intérieur d'un moteur MHEG, les "mh-objects" représentent les objets MHEG "disponibles". Les "mh-objects" sont des objets de "form 2" selon la description au 6.2.4 de l'ISO/CEI 13522-1 [5]. Un "mh-object" représente un objet MHEG, à savoir qu'il existe toujours un train binaire correspondant à un "mh-object". Un moteur MHEG ne devrait pas manipuler plus d'un "mh-object" pour la représentation d'un objet MHEG donné.

Il en résulte que les "mh-objects" manipulés par des moteurs MHEG peuvent être identifiés à l'aide d'identificateurs MHEG. De plus, d'autres mécanismes d'identification de "mh-objects" (par exemple l'identification symbolique) peuvent être définis par l'application, en supposant que leur représentation interne le permet. Ceci est plus particulièrement utile lorsque certains des objets MHEG, représentés par des "mh-objects" du moteur MHEG, ne sont pas identifiables, à savoir qu'ils n'ont pas d'identificateur MHEG. Ceci permet de garantir que tous les "mh-objects" sont identifiables.

Les "mh-objects" sont référencés de la même manière que les objets MHEG. Les références aux objets MHEG pour lesquels le moteur MHEG manipule un "mh-object" seront habituellement résolues en adressant ce "mh-object".

III.3 Rt-objects

Un "rt-object" est une instance (ou copie d') d'exécution d'un "mh-object" "modèle", créé et manipulé par un moteur MHEG à des fins de présentation. Un "rt-object" n'est pas un objet MHEG. A l'intérieur d'un moteur MHEG, les "rt-objects" représentent les objets MHEG rt- disponibles en cours d'exécution. Les "rt-objects" sont des objets de "form 3" selon la description au 6.2.4 de l'ISO/CEI 13522-1 [5]. Il peut y avoir zéro ou plusieurs "rt-objects" formant des copies "présentables" d'un "mh-object". Un "rt-object" a exactement un "mh-object" pour modèle.

Les "rt-objects" peuvent être identifiés à l'aide des identificateurs de "rt-objects" dont le composant "model object identification" est un identificateur MHEG. La structure et la représentation codée des identificateurs de "rt-objects" sont définies par la Rec. UIT-T T.172 (et l'ISO/CEI 13522-1) [5]. De plus, d'autres mécanismes d'identification de "rt-objects" (par exemple l'identification symbolique)

peuvent être définis par l'application, en supposant que leur représentation interne le permet. Ceci est plus particulièrement utile lorsque certains des objets MHEG, représentés par des "mh-objects" du moteur MHEG et utilisés comme modèles de "rt-objects", ne sont pas identifiables, à savoir qu'ils n'ont pas d'identificateur MHEG. Ceci permet de garantir que tous les "rt-objects" sont identifiables.

Les "rt-objects" peuvent être référencés à l'aide de références MHEG génériques.

III.4 Objets MHEG échangés

Les objets MHEG échangés sont des représentations d'objets MHEG communiquées en temps et lieu voulu à l'aide d'un réseau ou d'un média de stockage. Un objet MHEG donné (à savoir un train binaire) peut être échangé plusieurs fois entre plusieurs lieux différents, à savoir qu'il peut être représenté par de nombreux objets MHEG échangés. Un identificateur MHEG externe peut identifier un objet MHEG échangé et par conséquent, peut référencer un objet MHEG au travers de son emplacement et de son heure d'échange. Cependant, il faut noter qu'un identificateur MHEG externe n'a pas besoin réellement identifier un objet MHEG.

Les objets MHEG stockés sont des représentations d'objets MHEG habituellement situées dans des enregistrements de fichiers ou de bases de données. Par exemple, un objet MHEG donné (à savoir un train binaire) peut être stocké dans plusieurs endroits. De tels emplacements sont habituellement identifiés à l'aide de noms de fichiers ou d'identificateurs de bases de données. Un identificateur MHEG externe peut identifier l'endroit où un objet MHEG est stocké et par conséquent, référencer un objet MHEG au travers de son lieu de stockage.

APPENDICE IV

Principales caractéristiques de la SIR MHEG

La présente Recommandation a été développée afin de répondre à un nombre de besoins et de contraintes concernant:

– les applications utilisatrices;
– les fonctionnalités qu'elle fournit;
– son contexte d'utilisation par les applications;
– ses performances.

la SIR MHEG est dotée des attributs décrits dans le présent appendice.

IV.1 Caractéristiques des applications utilisatrices

La SIR MHEG satisfait les besoins des applications traitant:

1) de la manipulation d'entités MHEG;
2) des calculs, manipulations de variables et structures de contrôle;
3) du contrôle de dispositifs externes;
4) de l'acquisition de données;
5) de l'accès à des données externes;
6) de l'accès à des services d'exploitation externes arbitraires.

IV.1.1 Manipulation d'entités MHEG

Les applications manipulent habituellement des entités MHEG à des fins de présentations multimédias.

Cette caractéristique est mise en œuvre grâce à la fourniture des mécanismes décrits au IV.2.2.

IV.1.2 Calcul, manipulation de variables et contrôle de structure

Les applications ont besoin de ces fonctions de traitement de données pour implémenter des comportements dynamiques.

Ces fonctions sont assurées grâce à la fourniture des mécanismes décrits dans au IV.2.1.

IV.1.3 Contrôle de dispositifs externes

Les dispositifs fournissent souvent une interface spécifique à la plate-forme. Les composants de la plate-forme d'exécution qui implémentent une autre Recommandation et ses normes monomédias encapsulées supportent un nombre de dispositifs. De plus, certaines applications ont besoin de contrôler directement ces dispositifs ou d'autres dispositifs qui ne sont pas autrement supportés.

Cette fonction est assurée grâce à la fourniture des mécanismes décrits dans au IV.2.2.

NOTE – Dans ce contexte, un pilote de dispositif est un bloc service devant être fourni par l'environnement d'exécution.

IV.1.4 Acquisition de données

Les composants de la plate-forme d'exécution qui implémentent une autre Recommandation et ses normes monomédias encapsulées supportent un nombre de mécanismes d'acquisition de données. De plus, certaines applications ont besoin de contrôler directement l'acquisition de données (pour un réglage plus fin) à partir de ces mécanismes ou d'autres qui ne sont pas autrement supportés.

Cette fonction est assurée grâce à la fourniture des mécanismes décrits au IV.2.2.

NOTE – Dans ce contexte, les données acquises peuvent être récupérées par un "rt-script" via une réaction asynchrone aux messages de notification envoyés par le pilote du dispositif externe d'acquisition.

IV.1.5 Accès à des données externes

Certaines applications ont besoin d'accéder à des données non-MHEG devant être récupérées à partir de sources de stockage locales, de flux de données ou de dépôts de données distants à l'aide d'un composant de communication de l'environnement d'exécution.

Cette fonction est assurée grâce à la fourniture des mécanismes décrits au IV.2.2.

NOTE – Dans ce contexte, les données externes peuvent être récupérées au travers d'un composant système qui est un bloc service fourni par l'environnement d'exécution.

IV.1.6 Accès à des services externes d'exécution arbitraires

Les services d'exécution arbitraires font référence à tous les services dont la définition de l'interface n'est pas connue de l'implémentation avant l'échange de l'application.

Cette fonction est assurée grâce à la fourniture des mécanismes décrits au IV.2.2.

NOTE – Dans ce contexte, les fonctions de calcul externe peuvent être fournies par une bibliothèque ou un processus qui est un bloc service fourni par l'environnement d'exécution.

IV.2 Caractéristiques fonctionnelles

La SIR MHEG est utilisée pour exprimer:

1) des opérations de traitement de données (voir IV.2.1);
2) l'accès à des fonctions et données externes (voir IV.2.2).

IV.2.1 Opérations de traitement de données

Afin de supporter les opérations de traitement de données, la SIR MHEG fournit des mécanismes utilisés par les scripts échangés pour exprimer:

– la structure de types de données numériques construits et avancés;

– les variables et les valeurs de ces types;

– les instructions effectuant l'accès aux données et l'affectation des variables;

– les instructions ayant un effet sur le flux de contrôle d'exécution de script;

– les instructions effectuant les opérateurs arithmétiques, logiques et de comparaison.

Ces mécanismes sont décrits dans les paragraphes 8, 12 et 13.

IV.2.2 Accès à des fonctions et données externes

L'accès aux fonctions et données externes demande une coopération entre plusieurs composants du moteur MHEG-3 et son environnement d'exécution pour invoquer les fonctions, envoyer et recevoir les messages et échanger les données.

Pour supporter la manipulation des entités MHEG, c'est-à-dire l'accès aux données et fonctions MHEG, la SIR MHEG fournit aux scripts échangés des mécanismes pour:

– invoquer les actions MHEG élémentaires ;

– répondre aux actions MHEG destinées aux "rt-scripts";

– exprimer les valeurs et les variables des types de données MHEG.

Ces mécanismes sont décrits au paragraphe 11. Ils sont aussi utilisés pour exprimer l'échange de données et la synchronisation entre un "rt-script" et d'autres "rt-objects" ou entre les "rt-scripts" eux-mêmes.

Pour supporter la coopération avec l'environnement d'exécution (c'est-à-dire l'accès à des données et fonctions non MHEG), la SIR MHEG fournit aux scripts échangés des mécanismes pour:

– déclarer la structure des blocs fournis par l'environnement d'exécution;

– invoquer les services fournis par l'environnement d'exécution;

– réagir aux messages envoyés par l'environnement d'exécution;

– déclarer des types de données numériques construits et avancés;

– exprimer les variables et les valeurs de ces types.

Ces mécanismes sont décrits au paragraphe 10. Ils expriment aussi l'échange des données complexes et la synchronisation entre l'interpréteur de script SIR MHEG et les processus faisant partie de l'environnement d'exécution.

De plus, la présente Recommandation définit le mappage entre les déclarations de blocs SIR MHEG et les composants réels de l'environnement d'exécution. Ceci est réalisé en deux étapes:

– expression et utilisation d'une spécification d'interface abstraite à l'aide de la SIR MHEG. C'est le mécanisme IDL de mappage décrit au 14. Il contient les dispositions pour les scripts MHEG-3 échangés.

– mappage entre la spécification de l'interface abstraite et son implémentation à l'intérieur de l'environnement d'exécution d'un type de plate-forme donné. C'est le formulaire de spécification de mappage de plate-forme décrit dans l'Annexe D. Il contient les dispositions pour les implémentations de la norme MHEG-3.

IV.3 Caractéristiques techniques

La SIR MHEG satisfait aux caractéristiques techniques suivantes:

1) indépendance vis-à-vis du matériel;
2) représentation sous forme définitive;
3) compacité;
4) facilité d'implémentation;
5) efficacité d'interprétation;
6) ouverture et extensibilité;
7) caractère non révisable;
8) dispositions pour l'échange temps réel;
9) validation sémantique à des fins de qualité de service;
10) caractère vérifiable de la syntaxe (en relation avec les dangers de contamination);
11) représentation non propriétaire;
12) traitement de script sécurisé.

IV.3.1 Indépendance vis-à-vis du matériel

L'indépendance de la SIR MHEG vis-à-vis du logiciel, et par conséquence la portabilité des scripts échangés, est assurée, d'une part grâce à la définition d'un code de machine virtuelle utilisé pour exprimer les scripts échangés et, d'autre part grâce à la définition d'une machine virtuelle pour interpréter ce code. Seules des données typées sont utilisées. Il n'existe pas de contrainte sur la manière dont les données sont représentées ou manipulées en interne par les moteurs MHEG-3.

La représentation codée est basée sur les règles de codage ASN.1 qui sont indépendantes du matériel.

Les déclarations d'interfaces sont basées sur un mappage envers des spécifications abstraites d'interfaces pouvant être exprimées en IDL, ce qui est indépendant du matériel.

La capacité d'un composant de l'environnement d'exécution d'une plate-forme donnée à interopérer avec n'importe quel moteur MHEG-3 sur cette plate-forme est garantie grâce à l'utilisation de la spécification de mappage de plate-forme.

La capacité pour une implémentation de moteur MHEG-3 sur une plate-forme donnée à interopérer avec n'importe quel fournisseur de service à l'intérieur de l'environnement d'exécution est garantie grâce à l'utilisation de la spécification de mappage de plate-forme, en supposant que de tels fournisseurs de services soient conformes à cette spécification.

IV.3.2 Représentation sous forme définitive

La représentation sous forme définitive des scripts échangés est assurée par:

– l'utilisation de ASN.1 pour le codage des scripts échangés;
– un codage de machine virtuelle qui est sémantiquement proche d'une vaste classe d'ordinateurs généralistes;
– une architecture de machine virtuelle ayant un codage d'instruction efficace basé sur un mode d'adressage impliqué;
– un ordonnancement des déclarations réduisant la surcharge induite par le traitement des références en avant;
– le séquencement adéquat des instructions dans une routine.

IV.3.3 Compacité

La compacité de la représentation codée des scripts échangés est assurée grâce à plusieurs optimisations:

– la définition d'un code de machine virtuelle basé sur une machine à pile permet aux instructions de n'avoir que peu ou pas d'opérandes: la plus longue des instructions de la SIR MHEG tient sur 4 octets;

– l'utilisation d'un codage de type "chaîne d'octets" pour le code des routines permet de s'affranchir de l'overhead introduit par le TLV;

– les instructions codées dans les routines sont compactées, sans octets de bourrage.

– des constantes sont utilisées pour les déclarations de valeurs immédiates;

– des codes prédéfinis sont utilisés pour les types, opérations et messages MHEG;

– la déclaration d'une table de définitions de filets est utilisée pour optimiser l'expression du mappage entre les messages à destination du script et les routines ayant pour objectif de manipuler ces messages.

IV.3.4 Facilité d'implémentation

La facilité d'implémentation d'interpréteurs de script SIR MHEG est assurée grâce à:

– la définition d'un jeu d'instructions réduit;

– la définition précise d'une machine virtuelle;

– le nombre limité de concepts et d'identificateurs que les interpréteurs de script ont besoin de manipuler;

– la définition formelle de la sémantique des instructions.

IV.3.5 Efficacité dans l'interprétation

L'efficacité dans l'interprétation des scripts échangés par les interpréteurs de scripts SIR MHEG est assurée grâce à:

– l'utilisation d'un code de machine virtuelle à pile;

– l'utilisation d'instructions de bas niveau;

– l'utilisation d'une représentation sous forme définitive.

IV.3.6 Ouverture et extensibilité

L'ouverture et l'extensibilité de la SIR MHEG sont assurées par:

– une définition générique des interfaces pouvant être accédés à partir du code de script SIR MHEG;

– la capacité à accéder des objets MHEG et à invoquer des routines à partir d'un autre "rt-script";

– la possibilité d'ajouter de nouvelles instructions à la représentation sans modifier la structure des scripts échangés.

IV.3.7 Caractère non révisable

Le caractère non révisable est assuré grâce à l'utilisation d'une forme définitive et d'une représentation de bas niveau. Cette représentation est faite pour être produite par des outils informatiques spécialisés et ne permet pas de revenir facilement au code source d'origine réalisé par des opérateurs humains utilisant des langages de script et des environnements auteurs. Aussi, le risque d'altération indue de la sémantique du programme est-il limité.

IV.3.8 Dispositions pour l'échange temps-réel

La SIR MHEG s'inscrit dans le cadre de MHEG dont la structure générale a été réalisée pour satisfaire à des demandes d'échange temps réel.

La syntaxe des scripts échangés est définie de manière à optimiser la possibilité de traitement des déclarations qu'ils contiennent en la rendant possible "au fil de l'eau".

De plus, l'utilisation des règles de codage ASN.1 permet de détecter des erreurs lors de l'échange des scripts dans des environnements réseaux qui ne sont pas sûrs.

IV.3.9 Validation sémantique à des fins de qualité de service

A cause des contraintes sur la sémantique imposées par la présente Recommandation, le comportement d'un script échangé peut être testé pour en valider la performance avant son usage réel dans un contexte commercial. Les interpréteurs de script SIR MHEG peuvent être construits pour servir de référence pour tout ce qui concerne la manière dont les moteurs MHEG-3 conformes devraient se comporter lors de l'interprétation de scripts en phase de test échangés.

IV.3.10 Caractère vérifiable de la syntaxe (en relation avec les dangers de contamination)

La définition formelle de la syntaxe SIR MHEG peut être utilisée pour en vérifier le caractère correct et par conséquent, empêcher l'échange de code contenant des virus ayant pour objectif d'endommager le système hôte. Les implémentations peuvent décider d'effectuer des vérifications syntaxiques et sémantiques lors de l'exécution ou du chargement.

NOTE – La réalisation d'une SIR MHEG interprétée et dont la représentation est indépendante de la machine réduit les risques de contamination. Le risque résiduel peut provenir d'implémentations non certifiées ou incorrectes. La fourniture de mécanismes de cryptage, d'authentification ou d'autres mécanismes au niveau du transport est en dehors du domaine de la présente Recommandation.

IV.3.11 Représentation non propriétaire

La présente Recommandation suit la politique sur les droits de propriété intellectuelle des organismes internationaux de normalisation. Pour plus de détails, prière de se référer à la déclaration relative "aux droits de propriété intellectuelle" donnée en début de la présente Recommandation.

IV.3.12 Traitement de script sécurisé

Un réalisateur de système peut souhaiter assurer que l'exécution défectueuse de script, qu'elle soit intentionnelle ou accidentelle, ait des répercussions minimales sur le système de distribution, et que tout accès aux services externes puisse être surveillé avec soin. La machine virtuelle MHEG contient un nombre de caractéristiques soutenant cet objectif:

– interfaces explicites et fortement typées pour accéder aux services externes;
– jeu d'instructions fortement typé, de sorte que les opérations et les opérandes puissent être vérifiés avant traitement ou en cours d'exécution;
– pas d'adressage direct en mémoire (c'est-à-dire pas de pointeur arithmétique), empêchant ainsi la possibilité d'effets de bord;
– séparation des contextes de chaque objet "rt-script";
– séparation des contextes définis par chaque trame d'appel;
– pas de manipulation directe de pointeurs, d'identificateurs de données ou de types.

SERIES DES RECOMMANDATIONS UIT-T

Série A	Organisation du travail de l'UIT-T
Série B	Moyens d'expression: définitions, symboles, classification
Série C	Statistiques générales des télécommunications
Série D	Principes généraux de tarification
Série E	Exploitation générale du réseau, service téléphonique, exploitation des services et facteurs humains
Série F	Services de télécommunication non téléphoniques
Série G	Systèmes et supports de transmission, systèmes et réseaux numériques
Série H	Systèmes audiovisuels et multimédias
Série I	Réseau numérique à intégration de services
Série J	Transmission des signaux radiophoniques, télévisuels et autres signaux multimédias
Série K	Protection contre les perturbations
Série L	Construction, installation et protection des câbles et autres éléments des installations extérieures
Série M	Réseau de gestion des télécommunications et maintenance des réseaux: systèmes de transmission, de télégraphie, de télécopie, circuits téléphoniques et circuits loués internationaux
Série N	Maintenance: circuits internationaux de transmission radiophonique et télévisuelle
Série O	Spécifications des appareils de mesure
Série P	Qualité de transmission téléphonique, installations téléphoniques et réseaux locaux
Série Q	Commutation et signalisation
Série R	Transmission télégraphique
Série S	Equipements terminaux de télégraphie
Série T	**Terminaux des services télématiques**
Série U	Commutation télégraphique
Série V	Communications de données sur le réseau téléphonique
Série X	Réseaux pour données et communication entre systèmes ouverts
Série Z	Langages de programmation