THE Open GROUP

The Open Group Future Airborne Capability Environment (FACE™)

Shared Data Model Governance Plan, Edition 2.1

October 2015

Prepared by The Open Group FACE Consortium

NAVAIR Public Release 2014-696 (previous release 2014-275)

Future Airborne Capability Environment (FACE™):
Shared Data Model Governance Plan, Edition 2.1

Document Number: X1409US

Authored by The Open Group FACE Consortium.
Published by The Open Group, October 2015.

Comments relating to the material contained in this document may be submitted to:

The Open Group, 8 New England Executive Park, Burlington, MA 01803, United States

or by electronic mail to:

ogface-admin@opengroup.org

Contents

1. Introduction

1.1 Scope

This Governance Plan defines the policies, processes, and mechanisms governing the Future Airborne Capability Environment (FACE) Shared Data Model (SDM) and establishes the SDM Configuration Control Board (SDM CCB). The SDM CCB derives its authority from the FACE Steering Committee through a majority vote to approve this plan. The scope of responsibility of the SDM CCB is to manage change requests, new additions, and associated constraints to the FACE SDM, and document SDM updates associated with each edition of the FACE Technical Standard for Steering Committee approval, which will then be forwarded to The Open Group executive staff to approve publishing the most current version of the SDM. The plan also defines membership, roles, responsibilities, rules, and process flow for the SDM CCB.

1.2 Backward Compatibility

One of the guiding principles of management of the SDM is that no changes, updates, or deletions are allowed that would cause a previously certified Unit of Portability (UoP) to become non-conformant for the specific version of the standard to which that UoP was certified.

1.3 Editions

Particular considerations specific to each edition of the Technical Standard are documented in an appendix (e.g., Appendix A in the Shared Data Model Governance Plan, Edition 2.0 corresponds to FACE Technical Standard, Edition 2.0).

2. Governance Principles

To the extent possible, the FACE Data Architecture is governed by the application of "Rules of Construction". This promotes consistency between data elements created by different organizations over different time periods for different projects. Consistency in structure, specification of context, and data representations promotes reusability of Units of Portability (UoP). Additionally, rigorous Rules of Construction facilitate the automation of much of the conformance verification process.

2.1.1 Rules of Construction

The FACE Technical Standard specifies two types of Rules of Construction. The first is the FACE Metamodel. The Metamodel is based upon the Object Management Group (OMG) Essential Meta Object Facility (EMOF) standard and provides the specification for many of the Rules of Construction. Where the OMG EMOF is insufficient to capture the Rules of Construction, the OMG Object Constraint Language (OCL) standard is used to specify constraints on the EMOF specified model.

2.1.2 Governance Mechanism

The governance mechanism of the FACE Data Architecture consists of:

- FACE Metamodel
- FACE Object Constraint Language (OCL) Constraints
- Versioning
- Configuration Control Board (CCB)
- Conformance

The FACE Metamodel, established by the FACE Technical Standard and the FACE OCL Constraints are enforced by the FACE Conformance Program. Editions of the FACE Technical Standard define the FACE Metamodel applicable to that edition of the standard. The SDM CCB establishes the Conceptual Data Model Basis Elements, the Logical Data Model Basis Elements, and the Platform Data Model Basis Elements. Conformance verifies that each of these governance mechanisms is appropriately applied to a UoP Supplied Data Model (USM).

3. FACE SDM Versioning

The FACE Consortium will maintain two separate versions of the SDM. One will contain International Traffic in Arms Regulations (ITAR) restricted information and will be referred to as the ITAR SDM. The Public SDM is based on the ITAR SDM and contains the publicly releasable information. The Public SDM is a subset of the ITAR SDM. Updates to both the Public SDM and ITAR SDM are expected to occur no more than quarterly. The FACE Consortium will limit the differences between the ITAR SDM and Public SDM to information restricted by ITAR. See Section 7.2 for rationale on the SDM sets. The NAVAIR PAO Office, Patuxent River, MD is responsible for the ITAR publication release approval. The use of either the ITAR SDM or Public SDM is acceptable for achieving FACE conformance verification.

Additionally, there may be extensions to the ITAR SDM that are not releasable due to security restrictions. Any such models are referred to as ITAR SDM + extensions.

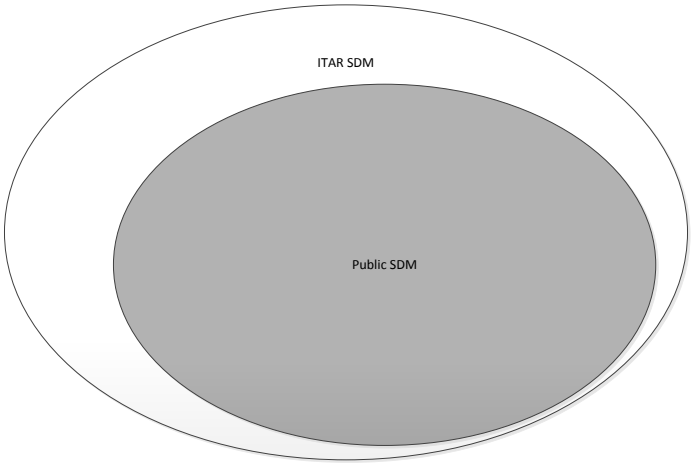


Figure 1: FACE SDMs

3.1 Major Editions

An ITAR SDM and Public SDM will be created to correspond to each edition of the FACE Technical Standard. These editions will be accessible through the FACE Reference Repository with corresponding access controls. The version identifier for the SDM will correspond to that of the FACE Technical Standard. Each of these versions will reside in the FACE SDM Reference Repository.

3.2 Minor Editions

The ITAR SDM and Public SDM for each edition of the FACE Technical Standard may be updated by the SDM CCB or as a result of new UoPs being verified through the conformance process. These updates will be accessible to FACE Consortium members immediately upon being loaded into the ITAR SDM. However, the new information will require government approval prior to public release. The government plans to approve the changes and release minor version updates to the Public SDM quarterly. Each of these editions will reside in the FACE SDM Reference Repository.

4. FACE Shared Data Model Configuration Control Board

4.1 Membership

The SDM Configuration Control Board (SDM CCB) official voting membership consists of the FACE Technical Working Group (TWG) Chair, FACE TWG Vice-Chair, Data Model Subcommittee Lead, Data Model Subcommittee Co-Lead, Transport Subcommittee Lead, and Transport Subcommittee Co-Lead. To support continued data model alignment activities, the FACE Steering Committee approved a single representative from a collaborating standards organization to be appointed as an SDM CCB voting member by their respective Alignment Advisory Group (e.g., UCS and FACE Consortium Alignment Advisory Group (UFCAAG), or others as appropriate).

In addition to official voting members, the following additional stakeholders shall receive communication of all significant SDM CCB events and have an opportunity to review and comment on all significant SDM CCB decisions and artifacts.

Stakeholders are the Subcommittee Lead or delegate for each of the FACE TWG and BWG Subcommittees. In the event one member of the SDM CCB holds more than one position stated above, the SDM CCB will appoint a substitute member from the list of Stakeholders.

- Current stakeholders from the FACE TWG:
  - General Enhancements Subcommittee Lead or delegate
  - Standard Subcommittee Lead or delegate
  - Configuration Subcommittee Lead or delegate
  - Conformance Verification Matrix Subcommittee Lead or delegate
  - Graphics Subcommittee Lead or delegate
  - Security Subcommittee Lead or delegate
  - Software Safety Subcommittee Lead or delegate
  - Reference Implementation Guide Subcommittee Lead or delegate
- Current stakeholders from the FACE Business Working Group:
  - Conformance Subcommittee Lead or delegate
  - Library Subcommittee Lead or delegate
  - Business Model Subcommittee Lead or delegate
  - Outreach Subcommittee Lead or delegate
- Current stakeholders from the Standards Alignment Advisory Group(s):
  - The appointed Advisory Group members from the FACE Consortium
  - The appointed Advisory Group members from the collaborating Standards Body(ies)

4.2 Roles and Responsibilities

4.2.1 SDM CCB Chair

The SDM CCB Chair shall be the FACE Data Model Subcommittee Lead. The SDM CCB Chair will be the primary facilitator of the SDM CCB performing the following duties:

- Retrieve Change Requests (CRs) and work with submitter(s) to clarify the CR
- Determine and set the length of the review time
- Initiate the review process described in Section 6
- Vote as described in Section 4.3

4.2.2 FACE SDM Administrator

The FACE SDM Administrator shall perform the following duties:

- Implement approved CR as quickly as possible
- Work with the SDM CCB to verify implementation

4.2.3 FACE SDM Configuration Manager

The FACE SDM Configuration Manager shall perform the following duties:

- Maintain configuration management of the FACE SDM
- Notify FACE TWG membership and FACE leadership of impending changes

4.2.4 Members

Members of the SDM CCB have the following duties:

- Vote on matters as they are brought forward
- Failure to vote within the time specified results in an Abstain vote being logged on the CR

4.3 Voting

Members of the SDM CCB vote using one of the following four choices:

- Approve
- Accept (with Modification)
- Reject (with Rationale)
- Abstain

A CR is approved if it has more Approve votes than Reject and Accept with Modification votes combined. If the request is not approved and there are Accept with Modification vote(s), a requestor may initiate a new CR vote based on the Modification(s) proposed. If there are multiple Accept with Modification votes they may be consolidated for subsequent votes or each submitted as a separate CR at the discretion of the requestor(s). A member of the SDM CCB may abstain from voting when their organization has an interest, either as a submitter or potential competitor, in the outcome of the CR that is

being considered by the SDM CCB. This would potentially alleviate any perception that their organization has an advantage due to an employee being a member of the SDM CCB. It allows the SDM CCB to maintain impartiality during the CR process. Preserving the impartiality of the SDM CCB is an important consideration as the CCB impacts the SDM which is a key component of the FACE Reference Architecture. A CR must gain 75% of the SDM CCB voting members' approval in order for the CR to pass.

The requester may withdraw a CR at any time prior to approval or rejection.

4.3.1 Voting Options

As noted above, there are four options:

- Approve
- Accept with Modification
- Reject (with Rationale)
- Abstain

The option to Accept with Modification must include details of the Modification proposed. The option to reject must include a rationale for rejection. Failure to provide adequate Modification or Rationale, as determined by the SDM CCB Chair, results in an Abstain vote.

4.3.2 Minority Reports

If a member votes differently than the majority (other than Abstain), the member has the option to write a Minority Report that will be included as an artifact of the CR.

5. FACE SDM Managed Elements

5.1 FACE OCL Constraints

In addition to the actual data elements, the constraints on the data model are also managed by the SDM CCB. Constraints shall be stored in file(s) that are external to the SDM and use the OMG OCL in accordance with the FACE Technical Standard.

5.2 SDM Elements

Any instance of a data model element may be deprecated by the SDM CCB. The use of elements that are flagged deprecated is discouraged. Deprecated elements are intended to be removed from the next future major version of the FACE SDM.

The Basis Elements defined by each edition of the FACE Technical Standard are located in their corresponding appendix (e.g., Appendix A in the Shared Data Model Governance Plan corresponds to FACE Technical Standard, Edition 2.0). These Basis Elements shall be managed by the SDM CCB.

An Extension Element is any element not in the appropriate SDM that has been created using the Basis Elements and the Rules of Construction. All Extension Elements must pass all conformance verification tests. Elements in this category provide new combinations of existing data elements or new views of data already represented in the SDM. These elements may be created by Software Suppliers in the development of their UoPs, and provided at the time of verification of conformance. The SDM CCB will review Extension Elements and add them, as appropriate, to the ITAR SDM (e.g., for potential reuse). In some cases the SDM CCB may make changes to the submitted CR that result in reuse of existing elements with new aliases as opposed to creation of new elements that duplicate the attributes of existing elements.

5.3 FACE Metamodel

Changes to the FACE Metamodel are not the responsibility of the SDM CCB but are managed as part of the release process of new editions of the FACE Technical Standard. They are therefore the responsibility of the FACE Data Model Subcommittee, the FACE TWG, and the FACE Steering Committee.

6. Change Request Process

6.1 Change Requests

Any person or organization may submit a Change Request (CR) to recommend FACE SDM additions or corrections. All CRs shall be submitted through the FACE Consortium PR/CR System and will follow the approved PR/CR process. The CR must include an attachment containing a valid FACE XMI file, that does not contain ITAR data, and includes all of the proposed additions to the FACE SDM.

A CR shall meet the EMOF metamodel specification from the relevant edition of the FACE Technical Standard. It shall also meet all of the OCL constraints for the relevant edition of the FACE Technical Standard or SDM Governance Plan. If not, the CR will be returned to be completed. Any SDM that incorporates the CR additions shall also meet the EMOF metamodel specification from the relevant edition of the FACE Technical Standard and all of the OCL constraints for the relevant edition of the FACE Technical Standard or SDM Governance Plan.

If the CR reaches the SDM CCB to be investigated, CRs will be evaluated and voted upon by CCB members and, if approved, will be implemented by the FACE SDM Administrator. The internal SDM CCB process is shown in Figure 2 and is described in more detail below.

Once the CR reaches the SDM CCB to be investigated, it is placed in an INITIATED state.

The CR is then reviewed by the SDM CCB and either accepted, returned, or rejected. The following steps are all done within the SDM CCB's purview:

- If the CR is valid, the CR is placed in the OPENED state and sent to the FACE SDM Administrator to implement.
- If the CR is returned due to a lack of information, the CR is placed in a RETURNED state and sent back to the user through the PR/CR process to request more information.
- If the CR is rejected because the change is deemed unattainable/invalid, the CR is placed in a REJECTED state and is sent back to the Submitter through the PR/CR process. Internal to the SDM CCB process, the CR is marked as CLOSED.

When the SDM is updated by the FACE SDM Administrator the CR is placed into a CORRECTED state. If the FACE SDM Administrator has issues with the CR, the CR would be placed into an EVALUATED state and sent back to the SDM CCB.

The SDM CCB verifies the update is correct.

- If the SDM CCB validates that the implementation is correct, the CR is placed in the CLOSED state and is returned to the PR/CR process to be completed.
- If the SDM CCB finds that the implementation is incorrect, the CR is placed back in an OPENED state and sent back to the FACE SDM Administrator.

After completing the internal SDM CCB process and the remaining steps of the PR/CR process, the updated FACE SDM is released.

6.2 Change Request State Definitions

INITIATED	CR has been placed into the tool and is awaiting review by the SDM CCB.
RETURNED	CR is not actionable and is sent back to the user for more information.
REJECTED	CR has been deemed unattainable/invalid and is considered the same as CLOSED as determined by the SDM CCB.
OPENED	CR has been provided to the FACE SDM Administrator for implementation into the SDM.
CORRECTED	CR has been completed by the FACE SDM Administrator.
EVALUATED	CR is not acceptable to the FACE SDM Administrator, and needs SDM CCB work.
CLOSED	SDM CCB has accepted the implementation – the CR returns to the PR/CR process to be completed.

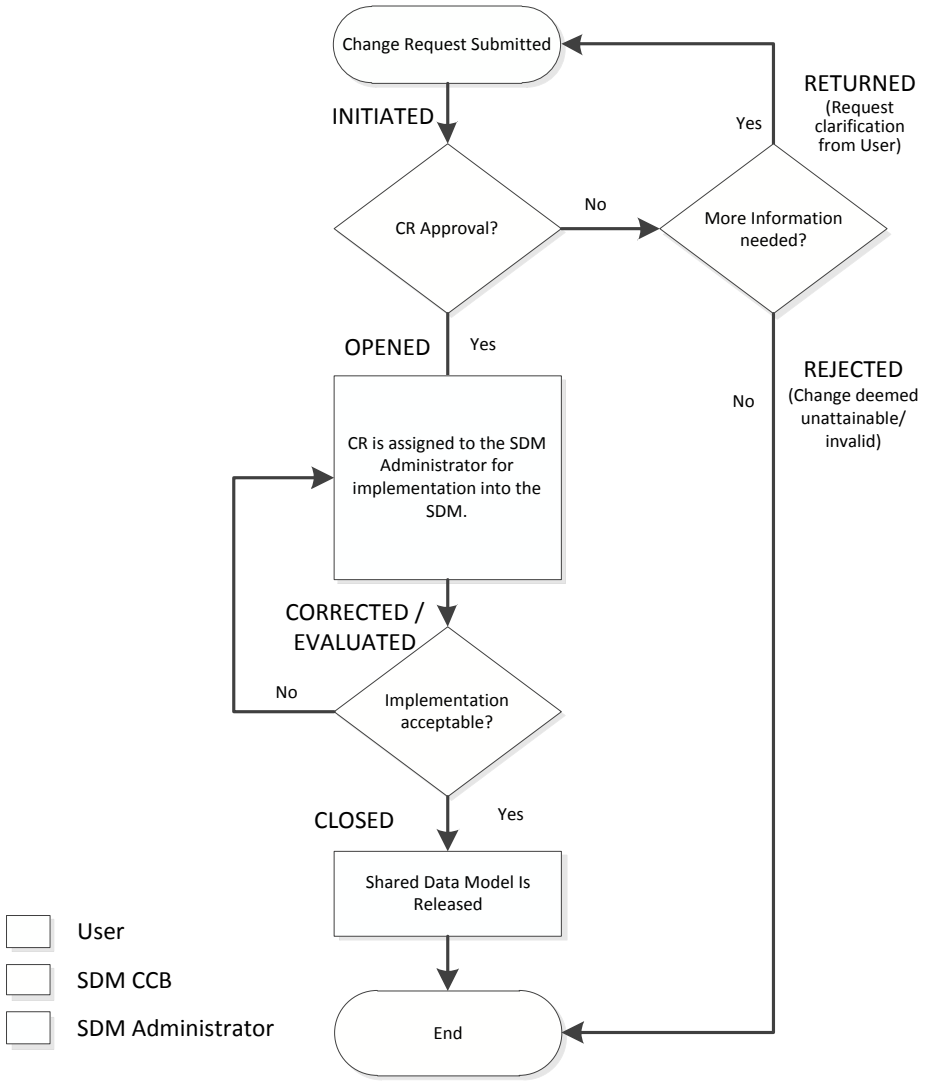


Figure 2: SDM Internal CR Process Flow

7. Data Model Conformance

A USM is verified for conformance to the FACE Metamodel as defined in the FACE Technical Standard. This involves meeting the requirements of the EMOF metamodel and the FACE OCL Constraints. Additionally, a USM is verified to be consistent with the appropriate SDM as part of the conformance verification process. Consistency is checked by comparing the USM elements to the appropriate SDM elements. The Software Supplier must state which SDM is to be used for conformance verification. The options are:

- ITAR SDM
- Public SDM
- ITAR SDM + extensions

For purposes of comparison, the following are definitions of additions and updates:

- Additions are elements in which the xmi:id of the element in the USM is not present in the SDM.
- Updates are elements in which the xmi:id of the element in the USM is present in the SDM and the element in the USM is not identical to the element in the SDM.

If a USM contains Basis Elements that are additions or updates, the USM will fail the conformance verification process. The FACE Conformance Test Suite verifies additions or updates to the Basis Elements defined in the relevant appendix. There are some Basis Elements that may be added and not cause conformance failure if the Software Supplier can provide evidence as to why the Extension Elements are not releasable. These Basis Elements are documented in the version-specific appendices.

Any additions to these types (in a USM) will be identified by the FACE Conformance Test Suite(s) and will show up as information in the resulting test report. They will trigger the Verification Authority to review the elements.

7.1 Additional SDM Types

In addition to the Basis Elements of the FACE SDM, other types in a USM will be checked against the SDM in the conformance process. The USM may have additional information but may not update existing elements in the SDM. Each element in a USM contains a unique identifier (xmi:id). This identifier will be used during the conformance verification process to determine whether an element in the SDM was renamed or modified.

7.2 Rationale

Some data types may be restricted from public release due to their sensitive nature. A review of FACE Technical Standard data types suggests that small subsets of metamodel types are the most likely candidates to prevent public release. A list of specific data types for each edition of the Technical Standard is presented in an appendix (e.g., Appendix A in the Shared Data Model Governance Plan, Edition 2.0 corresponds to FACE Technical Standard, Edition 2.0).

A FACE Technical Standard, Edition 2.0 Data Types

A review of FACE Technical Standard, Edition 2.0 data types suggests that the following data types are the most likely candidates to prevent public release:

- face.logical.FrameOfReference
- face.logical.Enumeration
- face.logical.SimpleMeasurement
- face.logical.CompositeMeasurement

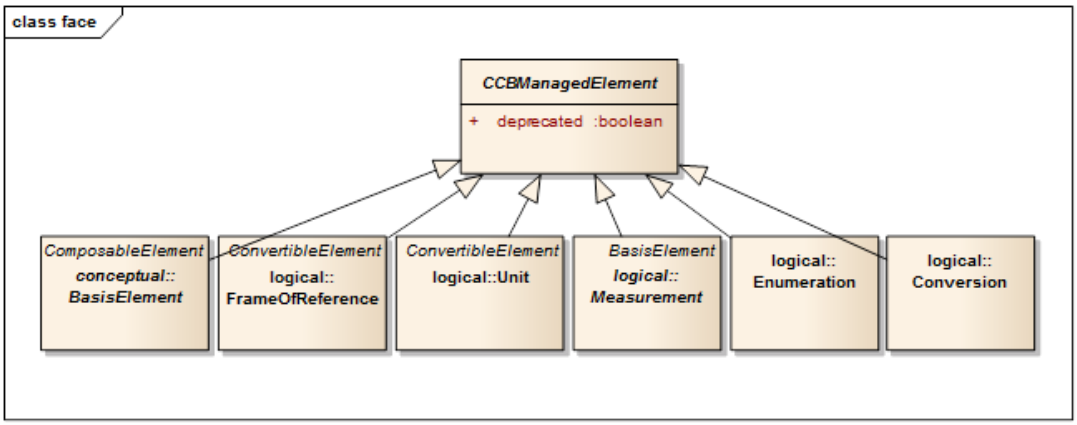
The following data type is also included as it is an abstract base type for face.logical.SimpleMeasurement and face.logical.CompositeMeasurement and could therefore contain sensitive information:

- face.logical.Measurement

A.1 Basis Elements

The following elements are the Basis Elements for FACE Edition 2.0:

- face.conceptual.Observable
- face.conceptual.InformationElement
- face.logical.Unit
- face.logical.Conversion (unit-to-unit conversions only)
- face.logical.Measurement (Abstract)
- face.logical.FrameOfReference
- face.logical.Enumeration
- face.logical.SimpleMeasurement
- face.logical.CompositeMeasurement



A.2 Constraints for *face* Package

```
package face

  context Element

    /*
     * 'tokenizeString' will return a sequence of strings that have been
     * split from the input string by the delimiter.
     * Assumes single character delimiter.
     */
    static def: tokenizeString(str : String, delimiter : String) : Sequence(String) =
      let i : Integer = str.indexOf(delimiter) in
      if i > 1
      then
          let token : String = str.substring(1, i-1) in
          if i = str.size()
          then
            Sequence{token}
          else
            let remainder : String = str.substring(i+1, str.size()) in
            let remainderTokens : Sequence(String) = tokenizeString(remainder,
                delimiter) in
            remainderTokens->prepend(token)
          endif
      else
        if i = str.size()
        then
          Sequence{}
        else
          if i = 1
          then
            let remainder : String = str.substring(i+1, str.size()) in
            self.tokenizeString(remainder, delimiter)
          else -- i <= 0
            Sequence{str}
          endif
        endif
      endif

endpackage

package face::conceptual
```

```
context Element
  /*
   * Every face::conceptual::Element in a FACE data model shall
   * have a unique name.
   */
  inv hasUniqueName:
    let conceptualElements: Set(face::conceptual::Element) =
      face::conceptual::Element.allInstances() in
    let otherConceptualElements: Set(face::conceptual::Element) =
      conceptualElements->excluding(self) in
    not otherConceptualElements->collect(name)->exists(e | e = self.name)

context Entity

  /*
   * A helper method that returns the Identity of an Entity
   */
def: getEntityIdentity() : Bag(OclAny) =
  let compositions : Bag(face::conceptual::Characteristic) =
    self->collect(composition)
  in

  let characteristics : Bag(face::conceptual::Characteristic) =
    if self.oclIsTypeOf(face::conceptual::Association)
    then
      self.oclAsType(face::conceptual::Association)->
        collect(associationEnd.oclAsType(face::conceptual::Characteristic))->
        union(compositions)
    else
      compositions
    endif
  in

    characteristics->collect(c | c.getIdentityContribution())

  /*
   * The full composition hierarchy of each conceptual Entity shall be
   * unique. Uniqueness is determined by number, multiplicity, and
   * type of its composed ComposableElement.
   */
inv entityIsUnique:
  let ceIdentity: Bag(OclAny) =
    self.getEntityIdentity() in
    face::conceptual::Entity.allInstances()->excluding(self)
      ->forAll(ce | ce.getEntityIdentity() <> ceIdentity
      )

  /*
   * Every face::conceptual::Entity in a FACE data model shall contain
   * a face::conceptual::InformationElement named 'UniqueIDType'.
   */
  inv hasUniqueID:
    self->collect(composition)->collect(type)
      ->exists(a | a.oclIsTypeOf(Observable)
        and a.name = 'UniqueIdentifier'
      )

  /*
   * Every face::conceptual::Composition within the scope
   * of an Entity must have a unique name.
   */
  inv allCompositionsHaveUniqueRolename:
    self->collect(composition)->isUnique(rolename)
```

```
  /*
   * A conceptual Entity must be composed from conceptual
   * BasisElements or other conceptual Entities which
   * are composed of conceptual BasisElements.
   */
  inv entityConstructed:
    let ceClosure = self->closure(ce |
      let types = ce->collect(composition)->collect(type)->asSet()
        in
      let entityTypes = types
        ->select(t | t.oclIsTypeOf(face::conceptual::Entity)) in
      entityTypes->collect(t | t.oclAsType(face::conceptual::Entity))
    ) in
    not ceClosure->includes(self)

context Association
  /*
   * Every face::conceptual::Characteristic within the scope
   * of an Association must have a unique name.
   */
  inv allCharacteristicsHaveUniqueRolename:
    let compositions =
      self->collect(composition)->
            collect(c|c.oclAsType(face::conceptual::Characteristic)) in
    let associatedEntities =
      self->collect(associationEnd)->
            collect(c|c.oclAsType(face::conceptual::Characteristic)) in
  let characteristics: Set(face::conceptual::Characteristic) =
    compositions->union(associatedEntities) in
  characteristics->isUnique(rolename)

context View

  /*
   * A helper method that returns the Identity of an Entity.
   */
def: getViewIdentity() : Bag(OclAny) =
  self->collect(characteristic)->collect(c | c.getIdentityContribution())

  /*
   * The each View shall be unique. Uniqueness is determined by
   * number, multiplicity, type, and context of its ProjectedCharacteristics.
   */
  inv viewIsUnique:
  let cvIdentity: Bag(OclAny) =
    self.getViewIdentity() in
    face::conceptual::View.allInstances()->excluding(self)
      ->forAll(cv | cv.getViewIdentity() <> cvIdentity
      )

  /*
   * The each View shall be unique. Uniqueness is determined by
   * number, multiplicity, type, and context of its ProjectedCharacteristics.
   */
  inv viewIsUnique2:
    true

context Characteristic

  /*
   * A helper method that returns the contribution that
   * a Characteristic makes to an Entity's identity.
```

```
   */
  def: getIdentityContribution() : Sequence(OclAny) =
  let type : face::conceptual::ComposableElement =
    if self.oclIsTypeOf(face::conceptual::Composition)
    then
      self.oclAsType(face::conceptual::Composition).type
    else
      self.oclAsType(face::conceptual::AssociatedEntity).type
    endif
  in

  Sequence{type, self.upperBound, self.lowerBound}

context Composition
  /*
   * For conceptual, logical, and platform Compositions the lowerBound
   * shall be less than or equal to upperBound.
   */
  inv lowerBound_LTE_UpperBound:
    self.upperBound <> -1 implies self.lowerBound <= self.upperBound
  /*
   * For conceptual, logical, and platform Compositions, upperBound shall
   * be == -1 or >= 1.
   */
  inv upperBoundValid:
    self.upperBound = -1 or self.upperBound >= 1
  /*
   * For conceptual, logical, and platform Compositions, lowerBound shall
   * be >= zero.
   */
  inv lowerBoundValid:
    self.lowerBound  >= 0

context CharacteristicProjection

  /*
   * A helper method that returns the contribution that
   * a Characteristic makes to an Entity's identity.
   */
  def: getIdentityContribution() : Sequence(OclAny) =
  Sequence{self.projectedCharacteristic, self.path}

  /*
   * Helper method to remove first node from a sequence of strings
   */
def: subSequenceStrings(s : Sequence(String)) : Sequence(String) =

  if s->size() = 1 then s->select(false) else s->subSequence(2, s->size()) endif

  /*
   * Helper method to determine if an association path resolution and subsequent
   * path selectors are valid from a relative location.
   */
def: associationPathResolutionValid(entity : face::conceptual::Entity, pathTokens :
  Sequence(String)) : Boolean =

  let currPathTokenSplit : Sequence(String) =
          Element::tokenizeString(pathTokens->first().replaceAll(']', ''),'[') in
  let currPathTokenRolename : String = currPathTokenSplit->first() in
  let currPathTokenAssociation : String = currPathTokenSplit->last() in
  let allAssociations : Collection(face::conceptual::Association) =
    face::conceptual::Association.allInstances() in
```

```
    -- there should be exactly one Association with expected name
    if (allAssociations->one(a | a.name = currPathTokenAssociation))
    then (
      let association : face::conceptual::Association =
        allAssociations->any(a | a.name = currPathTokenAssociation) in

      -- the Association should have exactly one AssociatedEntity
      -- with expected rolename
      if association.associationEnd->one(c | c.rolename = currPathTokenRolename)
      then (

        -- get the one AssociatedEntity with expected rolename
        let comp : face::conceptual::AssociatedEntity =
          association.associationEnd->any(c | c.rolename = currPathTokenRolename) in

        -- composition is valid if it exists, has a type, and the type is
        -- the current entity
        let compTypeValid : Boolean =
          comp <> null and comp.type <> null and comp.type = entity in

        -- path is valid it current node is valid and all subsequent nodes are valid
        compTypeValid and pathValid(association, subSequenceStrings(pathTokens))
      )
      else
        -- path is invalid if there is not exactly one AssociatedEntity
        -- with expected rolename
        false
      endif
    )
    else
      -- path is invalid if there is not exactly one Association
      -- with expected name
      false
    endif

    /*
     * Helper method to determine if a composition path resolution and subsequent
     * path selectors are valid from a relative location.
     */
  def: compositionPathResolutionValid(entity : face::conceptual::Entity, pathTokens :
    Sequence(String)) : Boolean =

    -- there should be exactly one Characteristic with expected rolename
    -- check if that Characteristic is a Composition
    if entity.composition->one(rolename = pathTokens->first())
    then (
      -- get composition with expected rolename
      let comp : face::conceptual::Composition =
        entity.composition->any(rolename = pathTokens->first()) in

      -- get associatedEntity with expected rolename
      let associatedEntity : face::conceptual::AssociatedEntity =
        entity.composition->any(rolename = pathTokens->first()) in

      -- path is valid it current node is valid and all subsequent nodes are valid
      self.pathValid(comp.type, subSequenceStrings(pathTokens))
    )
    else
      -- Check if the Characteristic with expected rolename is an AssociatedEntity
      if (entity.oclIsTypeOf(face::conceptual::Association) and
        entity.oclAsType(face::conceptual::Association).
          associationEnd->one(rolename = pathTokens->first())
      )
```

```
      then (
        -- get associatedEntity with expected rolename
        let associatedEntity : face::conceptual::AssociatedEntity =
          entity.oclAsType(face::conceptual::Association).
            associationEnd->any(rolename = pathTokens->first()) in

        -- path is valid it current node is valid and all subsequent nodes are valid
        self.pathValid(associatedEntity.type, subSequenceStrings(pathTokens))
)
      else
        -- path is invalid if there is not exactly
        -- one Characteristic with expected rolename
        false
      endif
    endif

    /*
     * Helper method to determine if a path relative to a ComposableElement is valid.
     */
  def: pathValid(ce : face::conceptual::ComposableElement, pathTokens :
    Sequence(String)) : Boolean =

    if (ce = null)
    then
      -- if ce is null then the path is invalid
      false
    else
      if pathTokens->size() = 0
      then
        -- if there are no more path tokens then the projection is valid
        true

      else (
        -- more path tokens indicate this must be an entity
        if ce.oclIsKindOf(face::conceptual::Entity)
        then (
          let entity : face::conceptual::Entity =
            ce.oclAsType(face::conceptual::Entity) in

          -- association resolutions have a square brackets
          if pathTokens->first().indexOf('[') > 0
          then
            associationPathResolutionValid(entity, pathTokens)
          else
            compositionPathResolutionValid(entity, pathTokens)
          endif
        )
        else (
          -- more path tokens indicate this must be an entity,
          -- if not then the path is invalid
          false
        )
        endif
      )
      endif
    endif

    /*
     * CharacteristicProjection must have a valid path.
    */
  inv characteristicProjectionPathValid:
    let ce : face::conceptual::ComposableElement =
      self.projectedCharacteristic.oclAsType(face::conceptual::ComposableElement) in
```

```
    let tokens : Sequence(String) =
      Element::tokenizeString(self.path.replaceAll('->', '.'), '.') in
    self.pathValid(ce, tokens)

    /*
     * CharacteristicProjection must have a valid path format.
     */
  inv characteristicProjectionValidFormat:

    self.path <> null implies

    --remove all entity projection substrings (e.g., ".description")
    let pathTmp1 = self.path.replaceAll('\\.[_a-zA-Z0-9]+', '') in

    --remove all entity projection substrings (e.g., "->description[A1]")
    let pathTmp2 = pathTmp1.replaceAll('->[_a-zA-Z0-9]+\\[[_a-zA-Z0-9]+\\]', '') in

    pathTmp2.size() = 0

  context InformationElement
    /*
     * Information Elements will be deprecated in future versions of FACE.
     */
    inv informationElementDepricated:
      false

endpackage
```

A.3 Constraints for *face::logical* Package

```
package face::logical

  context Element
    /*
     * Every face::logical::Element except constraints in a FACE data model shall
     * have a unique name.
     */
    inv hasUniqueName:
      if self.oclIsKindOf(face::logical::Constraint)
      then
          -- do not check name on Constraint specification
          true
    else
        -- get all instances of face::logical::Element
        let logicalElements: Set(face::logical::Element) =
          face::logical::Element.allInstances() in
        let otherLogicalElements: Set(face::logical::Element) =
          logicalElements->excluding(self) in
        not otherLogicalElements->collect(name)->exists(e | e = self.name)
    endif

  context Entity
    /*
     * Every face::logical::Composition within the scope
     * of an Entity must have a unique name.
     */
    inv allCompositionsHaveUniqueRolename:
      self->collect(composition)->isUnique(rolename)

  context Association
    /*
```

```
   * Every face::logical::Characteristic within the scope
   * of an Association must have a unique name.
   */
  inv allCharacteristicsHaveUniqueRolename:
    let compositions =
      self->collect(composition)->
            collect(c|c.oclAsType(face::logical::Characteristic)) in
    let associatedEntities =
      self->collect(associatedEntity)->
            collect(c|c.oclAsType(face::logical::Characteristic)) in
  let characteristics: Set(face::logical::Characteristic) =
    compositions->union(associatedEntities) in
  characteristics->isUnique(rolename)

context Composition
/*
 * Ensure that when an element realizes another element, the
 * upper and lower bounds of the realized entity match those
 * of the realizing entity.
 */
inv logicalLowerBoundEqualsConceptual:
  let realizedComposition: face::conceptual::Composition = self.realizes in
  self.lowerBound = realizedComposition.lowerBound

inv logicalUpperBoundEqualsConceptual:
  let realizedComposition: face::conceptual::Composition = self.realizes in
  self.upperBound = realizedComposition.upperBound

/* A logical entity composition hierarchy must be consistent
 * with the composition hierarchy of the conceptual entity
 * that it realizes. The logical measurements must correspond
 * with the conceptual observables.
 */
inv logicalEntityConsistentWithConceptual:
  let realizedComposition: face::conceptual::Composition = self.realizes in
  let type: face::logical::ComposableElement = self.type in
  if type = null then
    false
  else
  if type.oclIsKindOf(face::logical::Entity) then
    let entityType: face::logical::Entity = type.oclAsType(face::logical::Entity) in
    let conceptualType: face::conceptual::Entity = entityType.realizes in
    conceptualType = realizedComposition.type
  else
  if type.oclIsKindOf(face::logical::Measurement) then
    let measurementType: face::logical::Measurement =
        type.oclAsType(face::logical::Measurement) in
    let conceptualType: face::conceptual::Observable = measurementType.realizes in
    conceptualType = realizedComposition.type
  else
  if type.oclIsKindOf(face::logical::InformationElement) then
    let ieType: face::logical::InformationElement =
        type.oclAsType(face::logical::InformationElement) in
    let conceptualType: face::conceptual::InformationElement = ieType.realizes in
    conceptualType = realizedComposition.type
  else
    false
  endif
  endif
  endif
  endif

context CharacteristicProjection
```

```
    /*
     * CharacteristicProjection must have a valid path format.
     */
  inv characteristicProjectionValidFormat:

    self.path <> null implies

    --remove all entity projection substrings (e.g. ".description")
    let pathTmp1 = self.path.replaceAll('\\.[_a-zA-Z0-9]+', '') in

    --remove all entity projection substrings (e.g. "->description[A1]")
    let pathTmp2 = pathTmp1.replaceAll('->[_a-zA-Z0-9]+\\[[_a-zA-Z0-9]+\\]', '') in

    pathTmp2.size() = 0

  context Conversion
    /*
     * A face::logical::Conversion shall associate two
     * face::logical::ConvertableElement that are of the same metaclass
     * (e.g., only unit to unit or FrameOfReference to
     * FrameOfReference conversions are allowed).
     */
    inv unitConvertsToUnit:
      self.source.oclIsTypeOf(Unit) implies
        self.destination.oclIsTypeOf(Unit)
    inv frameOfReferenceConvertsToFrameOfReference:
      self.source.oclIsTypeOf(FrameOfReference) implies
        self.destination.oclIsTypeOf(FrameOfReference)

  context InformationElement
    /*
     * Information Elements will be deprecated in future versions of FACE.
     */
    inv informationElementDepricated:
      false

endpackage
```

A.4 Constraints for *face::platform* Package

```
package face::platform

  context Element
    /*
     * Every face::platform::Element in a FACE data model shall
     * have a unique name.
     */
    inv hasUniqueName:
      let platformElements: Set(face::platform::Element) =
        face::platform::Element.allInstances() in
      let otherPlatformElements: Set(face::platform::Element) =
        platformElements->excluding(self) in
      not otherPlatformElements->collect(name)->exists(e | e = self.name)

  context Composition
  /*
   * Ensure that when an element realizes another element, the
   * upper and lower bounds of the realized entity match those
   * of the realizing entity.
   */
  inv platformLowerBoundEqualsLogical:
```

```
  let realizedLogicalComposition: face::logical::Composition = self.realizes in
  self.lowerBound = realizedLogicalComposition.lowerBound

inv platformUpperBoundEqualsLogical:
  let realizedLogicalComposition: face::logical::Composition = self.realizes in
  self.upperBound = realizedLogicalComposition.upperBound

/* A platform entity composition hierarchy must be consistent
 * with the composition hierarchy of the logical entity
 * that it realizes. The platform value types must correspond
 * with the logical measurements and information elements.
 */
inv platformEntityConsistentWithLogical:
  let realizedComposition: face::logical::Composition = self.realizes in
  let type: face::platform::ComposableElement = self.type in
  if type = null then
    false
  else
  if type.oclIsKindOf(face::platform::Entity) then
    let entityType: face::platform::Entity =
        type.oclAsType(face::platform::Entity) in
    let logicalType: face::logical::Entity = entityType.realizes in
    logicalType = realizedComposition.type
  else
  if type.oclIsKindOf(face::platform::IDLPrimitive) then
    let idlType: face::platform::IDLPrimitive =
        type.oclAsType(face::platform::IDLPrimitive) in
    let logicalType: face::logical::ValueElement = idlType.realizes in
    logicalType = realizedComposition.type
  else
  if type.oclIsKindOf(face::platform::IDLStruct) then
    let idlType: face::platform::IDLStruct =
        type.oclAsType(face::platform::IDLStruct) in
    let logicalType: face::logical::ValueElement = idlType.realizes in
    logicalType = realizedComposition.type
  else
    false
  endif
  endif
  endif
  endif

/*
 * Ensure that composition rolename does not conflict with
 * a reserved word in IDL or FACE supported programming
 * language.
 */

inv compositionNameNotReservedWord:
    let name: String = self.rolename.toLower() in
  name <> 'abstract' and
  name <> 'any' and
  name <> 'attribute' and
  name <> 'boolean' and
  name <> 'case' and
  name <> 'char' and
  name <> 'component' and
  name <> 'const' and
  name <> 'consumes' and
  name <> 'context' and
  name <> 'custom' and
  name <> 'default' and
  name <> 'double' and
```

```
    name <> 'emits' and
    name <> 'enum' and
    name <> 'eventtype' and
    name <> 'exception' and
    name <> 'factory' and
    name <> 'false' and
    name <> 'finder' and
    name <> 'fixed' and
    name <> 'float' and
    name <> 'getraises' and
    name <> 'home' and
    name <> 'import' and
    name <> 'in' and
    name <> 'inout' and
    name <> 'interface' and
    name <> 'local' and
    name <> 'long' and
    name <> 'manages' and
    name <> 'module' and
    name <> 'multiple' and
    name <> 'native' and
    name <> 'object' and
    name <> 'octet' and
    name <> 'oneway' and
    name <> 'out' and
    name <> 'primarykey' and
    name <> 'private' and
    name <> 'provides' and
    name <> 'public' and
    name <> 'publishes' and
    name <> 'raises' and
    name <> 'readonly' and
    name <> 'sequence' and
    name <> 'setraises' and
    name <> 'short' and
    name <> 'string' and
    name <> 'struct' and
    name <> 'supports' and
    name <> 'switch' and
    name <> 'true' and
    name <> 'truncatable' and
    name <> 'typedef' and
    name <> 'typeid' and
    name <> 'typeprefix' and
    name <> 'union' and
    name <> 'unsigned' and
    name <> 'uses' and
    name <> 'valuebase' and
    name <> 'valuetype' and
    name <> 'void' and
    name <> 'wchar' and
    name <> 'wstring'

  context Entity

    /*
     * Every face::platform::Composition within the scope
     * of an Entity must have a unique name.
     */
    inv allCompositionsHaveUniqueRolename:
      self->collect(composition)->isUnique(rolename)

  /*
```

```
* Ensure that entity name does not conflict with
* a reserved word in IDL or FACE supported programming
* language.
*/

 inv entityNameNotReservedWord:
   let name: String = self.name.toLower() in
 name <> 'abstract' and
 name <> 'any' and
 name <> 'attribute' and
 name <> 'boolean' and
 name <> 'case' and
 name <> 'char' and
 name <> 'component' and
 name <> 'const' and
 name <> 'consumes' and
 name <> 'context' and
 name <> 'custom' and
 name <> 'default' and
 name <> 'double' and
 name <> 'emits' and
 name <> 'enum' and
 name <> 'eventtype' and
 name <> 'exception' and
 name <> 'factory' and
 name <> 'false' and
 name <> 'finder' and
 name <> 'fixed' and
 name <> 'float' and
 name <> 'getraises' and
 name <> 'home' and
 name <> 'import' and
 name <> 'in' and
 name <> 'inout' and
 name <> 'interface' and
 name <> 'local' and
 name <> 'long' and
 name <> 'manages' and
 name <> 'module' and
 name <> 'multiple' and
 name <> 'native' and
 name <> 'object' and
 name <> 'octet' and
 name <> 'oneway' and
 name <> 'out' and
 name <> 'primarykey' and
 name <> 'private' and
 name <> 'provides' and
 name <> 'public' and
 name <> 'publishes' and
 name <> 'raises' and
 name <> 'readonly' and
 name <> 'sequence' and
 name <> 'setraises' and
 name <> 'short' and
 name <> 'string' and
 name <> 'struct' and
 name <> 'supports' and
 name <> 'switch' and
 name <> 'true' and
 name <> 'truncatable' and
 name <> 'typedef' and
 name <> 'typeid' and
```

```
  name <> 'typeprefix' and
  name <> 'union' and
  name <> 'unsigned' and
  name <> 'uses' and
  name <> 'valuebase' and
  name <> 'valuetype' and
  name <> 'void' and
  name <> 'wchar' and
  name <> 'wstring'

context Association
  /*
   * Every face::logical::Characteristic within the scope
   * of an Association must have a unique name.
   */
  inv allCharacteristicsHaveUniqueRolename:
    let compositions =
      self->collect(composition)->
            collect(c|c.oclAsType(face::platform::Characteristic)) in
    let associatedEntities =
      self->collect(associatedEntity)->
            collect(c|c.oclAsType(face::platform::Characteristic)) in
  let characteristics: Set(face::platform::Characteristic) =
    compositions->union(associatedEntities) in
  characteristics->isUnique(rolename)

context View

/*
 * Ensure that view name does not conflict with
 * a reserved word in IDL or FACE supported programming
 * language.
 */

  inv viewNameNotReservedWord:
    let name: String = self.name.toLower() in
  name <> 'abstract' and
  name <> 'any' and
  name <> 'attribute' and
  name <> 'boolean' and
  name <> 'case' and
  name <> 'char' and
  name <> 'component' and
  name <> 'const' and
  name <> 'consumes' and
  name <> 'context' and
  name <> 'custom' and
  name <> 'default' and
  name <> 'double' and
  name <> 'emits' and
  name <> 'enum' and
  name <> 'eventtype' and
  name <> 'exception' and
  name <> 'factory' and
  name <> 'false' and
  name <> 'finder' and
  name <> 'fixed' and
  name <> 'float' and
  name <> 'getraises' and
  name <> 'home' and
  name <> 'import' and
  name <> 'in' and
  name <> 'inout' and
```

```
    name <> 'interface' and
    name <> 'local' and
    name <> 'long' and
    name <> 'manages' and
    name <> 'module' and
    name <> 'multiple' and
    name <> 'native' and
    name <> 'object' and
    name <> 'octet' and
    name <> 'oneway' and
    name <> 'out' and
    name <> 'primarykey' and
    name <> 'private' and
    name <> 'provides' and
    name <> 'public' and
    name <> 'publishes' and
    name <> 'raises' and
    name <> 'readonly' and
    name <> 'sequence' and
    name <> 'setraises' and
    name <> 'short' and
    name <> 'string' and
    name <> 'struct' and
    name <> 'supports' and
    name <> 'switch' and
    name <> 'true' and
    name <> 'truncatable' and
    name <> 'typedef' and
    name <> 'typeid' and
    name <> 'typeprefix' and
    name <> 'union' and
    name <> 'unsigned' and
    name <> 'uses' and
    name <> 'valuebase' and
    name <> 'valuetype' and
    name <> 'void' and
    name <> 'wchar' and
    name <> 'wstring'

  context CharacteristicProjection

  /*
   * CharacteristicProjection must have a valid path format.
   */
  inv characteristicProjectionValidFormat:

    self.path <> null implies

    --remove all entity projection substrings (e.g. ".description")
    let pathTmp1 = self.path.replaceAll('\\.[_a-zA-Z0-9]+', '') in

    --remove all entity projection substrings (e.g. "->description[A1]")
    let pathTmp2 = pathTmp1.replaceAll('->[_a-zA-Z0-9]+\\[[_a-zA-Z0-9]+\\]', '') in

    pathTmp2.size() = 0

    /*
     * Ensure that characteristic projection rolename does
     * not conflict with a reserved word in IDL or FACE
     * supported programming language.
     */

    inv characteristicProjectionNameNotReservedWord:
```

```
  let name: String = self.rolename.toLower() in
name <> 'abstract' and
name <> 'any' and
name <> 'attribute' and
name <> 'boolean' and
name <> 'case' and
name <> 'char' and
name <> 'component' and
name <> 'const' and
name <> 'consumes' and
name <> 'context' and
name <> 'custom' and
name <> 'default' and
name <> 'double' and
name <> 'emits' and
name <> 'enum' and
name <> 'eventtype' and
name <> 'exception' and
name <> 'factory' and
name <> 'false' and
name <> 'finder' and
name <> 'fixed' and
name <> 'float' and
name <> 'getraises' and
name <> 'home' and
name <> 'import' and
name <> 'in' and
name <> 'inout' and
name <> 'interface' and
name <> 'local' and
name <> 'long' and
name <> 'manages' and
name <> 'module' and
name <> 'multiple' and
name <> 'native' and
name <> 'object' and
name <> 'octet' and
name <> 'oneway' and
name <> 'out' and
name <> 'primarykey' and
name <> 'private' and
name <> 'provides' and
name <> 'public' and
name <> 'publishes' and
name <> 'raises' and
name <> 'readonly' and
name <> 'sequence' and
name <> 'setraises' and
name <> 'short' and
name <> 'string' and
name <> 'struct' and
name <> 'supports' and
name <> 'switch' and
name <> 'true' and
name <> 'truncatable' and
name <> 'typedef' and
name <> 'typeid' and
name <> 'typeprefix' and
name <> 'union' and
name <> 'unsigned' and
name <> 'uses' and
name <> 'valuebase' and
name <> 'valuetype' and
```

```
    name <> 'void' and
    name <> 'wchar' and
    name <> 'wstring'

  context IDLComposition
    /* A platform idl struct composition hierarchy must be consistent
     * with the composition hierarchy of the logical element
     * that it realizes. The platform value types must correspond
     * with the logical measurements and information elements.
     */
    inv idlStructConsistentWithLogical:
      let realizedComposition: face::logical::MeasurementComposition =
          self.realizes in
      let type: face::platform::ComposableElement = self.type in
      if type = null then
        false
      else
      if type.oclIsKindOf(face::platform::IDLPrimitive) then
        let idlType: face::platform::IDLPrimitive =
            type.oclAsType(face::platform::IDLPrimitive) in
        let logicalType: face::logical::ValueElement = idlType.realizes in
        logicalType = realizedComposition.type
      else
      if type.oclIsKindOf(face::platform::IDLStruct) then
        let idlType: face::platform::IDLStruct =
            type.oclAsType(face::platform::IDLStruct) in
        let logicalType: face::logical::ValueElement = idlType.realizes in
        logicalType = realizedComposition.type
      else
        false
      endif
      endif
      endif

endpackage
```

A.5 Constraints for *face::uop* Package

```
package face::uop

  context Alias

  /*
   * Ensure that alias name does not conflict with
   * a reserved word in IDL or FACE supported programming
   * language.
   */

    inv aliasNameNotReservedWord:
    let name: String = self.name.toLower() in
    name <> 'abstract' and
    name <> 'any' and
    name <> 'attribute' and
    name <> 'boolean' and
    name <> 'case' and
    name <> 'char' and
    name <> 'component' and
    name <> 'const' and
    name <> 'consumes' and
    name <> 'context' and
    name <> 'custom' and
```

```
    name <> 'default' and
    name <> 'double' and
    name <> 'emits' and
    name <> 'enum' and
    name <> 'eventtype' and
    name <> 'exception' and
    name <> 'factory' and
    name <> 'false' and
    name <> 'finder' and
    name <> 'fixed' and
    name <> 'float' and
    name <> 'getraises' and
    name <> 'home' and
    name <> 'import' and
    name <> 'in' and
    name <> 'inout' and
    name <> 'interface' and
    name <> 'local' and
    name <> 'long' and
    name <> 'manages' and
    name <> 'module' and
    name <> 'multiple' and
    name <> 'native' and
    name <> 'object' and
    name <> 'octet' and
    name <> 'oneway' and
    name <> 'out' and
    name <> 'primarykey' and
    name <> 'private' and
    name <> 'provides' and
    name <> 'public' and
    name <> 'publishes' and
    name <> 'raises' and
    name <> 'readonly' and
    name <> 'sequence' and
    name <> 'setraises' and
    name <> 'short' and
    name <> 'string' and
    name <> 'struct' and
    name <> 'supports' and
    name <> 'switch' and
    name <> 'true' and
    name <> 'truncatable' and
    name <> 'typedef' and
    name <> 'typeid' and
    name <> 'typeprefix' and
    name <> 'union' and
    name <> 'unsigned' and
    name <> 'uses' and
    name <> 'valuebase' and
    name <> 'valuetype' and
    name <> 'void' and
    name <> 'wchar' and
    name <> 'wstring'

endpackage
```

B FACE Technical Standard, Edition 2.1 Data Types

A review of FACE Technical Standard, Edition 2.1 data types suggests that the following data types are the most likely candidates to prevent public release:

- face.logical.Landmark
- face.logical.ReferencePoint
- face.logical.ReferencePointPart
- face.logical.MeasurementSystem
- face.logical.MeasurementSystemAxis
- face.logical.CoordinateSystem
- face.logical.CoordinateSystemAxis

B.1 Basis Elements

The following elements are the Basis Elements for FACE Edition 2.1:

- face.conceptual.Observable
- face.logical.Unit
- face.logical.Landmark
- face.logical.ReferencePoint
- face.logical.ReferencePointPart
- face.logical.MeasurementSystem
- face.logical.MeasurementSystemAxis
- face.logical.CoordinateSystem
- face.logical.CoordinateSystemAxis
- face.logical.MeasurementSystemConversion
- face.logical.Boolean
- face.logical.Character
- face.logical.Numeric
- face.logical.Integer
- face.logical.Natural
- face.logical.NonNegativeReal

- face.logical.Real
- face.logical.String

B.2 Constraint Helper Methods

```
package face

  context Element
    /*
     * 'tokenizeString' will return a sequence of strings that have been
     * split from the input string by the delimiter.
     * Assumes single character delimiter.
     */
    static def: tokenizeString(str : String, delimiter : String) : Sequence(String) =
      let i : Integer = str.indexOf(delimiter) in
      if i > 1
      then
          let token : String = str.substring(1, i-1) in
          if i = str.size()
          then
            Sequence{token}
          else
            let remainder : String = str.substring(i+1, str.size()) in
            let remainderTokens : Sequence(String) = tokenizeString(remainder,
              delimiter) in
            remainderTokens->prepend(token)
          endif
      else
        if i = str.size()
        then
          Sequence{}
        else
          if i = 1
          then
            let remainder : String = str.substring(i+1, str.size()) in
            self.tokenizeString(remainder, delimiter)
          else -- i <= 0
            Sequence{str}
          endif
        endif
      endif

    /*
     * Helper method to remove first node from a sequence of strings.
     */
    static def: removeFirstString(s : Sequence(String)) : Sequence(String) =
      if s->size() = 1 then s->select(false) else s->subSequence(2, s->size()) endif

    /*
     * Helper method to determine if string is a valid identifier.
     */
    static def: isValidIdentifier(str : String) : Boolean =
      str.size() > 0 and
      str.replaceAll('[_a-zA-Z][_a-zA-Z0-9]*', '').size() = 0 and
      not isReservedWord(str)

    /*
     * Helper method to determine if string is an valid format for a path.
     */
    static def: isValidPathFormat(str : String) : Boolean =
```

```
    str <> null implies

    --remove all entity projection substrings (e.g., ".description")
    let pathTmp1 = str.replaceAll('\\.[_a-zA-Z0-9]+', '') in

    --remove all entity projection substrings (e.g., "->description[A1]")
    let pathTmp2 = pathTmp1.replaceAll('->[_a-zA-Z0-9]+\\[[_a-zA-Z0-9]+\\]', '') in

    pathTmp2.size() = 0

  /*
   * Helper method to determine if string is an IDL reserved word.
   */
  static def: isReservedWord(str : String) : Boolean =
    let strLower: String = str.toLower() in
      strLower = 'abstract' or
      strLower = 'any' or
      strLower = 'attribute' or
      strLower = 'boolean' or
      strLower = 'case' or
      strLower = 'char' or
      strLower = 'component' or
      strLower = 'const' or
      strLower = 'consumes' or
      strLower = 'context' or
      strLower = 'custom' or
      strLower = 'default' or
      strLower = 'double' or
      strLower = 'emits' or
      strLower = 'enum' or
      strLower = 'eventtype' or
      strLower = 'exception' or
      strLower = 'factory' or
      strLower = 'false' or
      strLower = 'finder' or
      strLower = 'fixed' or
      strLower = 'float' or
      strLower = 'getraises' or
      strLower = 'home' or
      strLower = 'import' or
      strLower = 'in' or
      strLower = 'inout' or
      strLower = 'interface' or
      strLower = 'local' or
      strLower = 'long' or
      strLower = 'manages' or
      strLower = 'module' or
      strLower = 'multiple' or
      strLower = 'native' or
      strLower = 'object' or
      strLower = 'octet' or
      strLower = 'oneway' or
      strLower = 'out' or
      strLower = 'primarykey' or
      strLower = 'private' or
      strLower = 'provides' or
      strLower = 'public' or
      strLower = 'publishes' or
      strLower = 'raises' or
      strLower = 'readonly' or
      strLower = 'sequence' or
      strLower = 'setraises' or
      strLower = 'short' or
```

```
        strLower = 'string' or
        strLower = 'struct' or
        strLower = 'supports' or
        strLower = 'switch' or
        strLower = 'true' or
        strLower = 'truncatable' or
        strLower = 'typedef' or
        strLower = 'typeid' or
        strLower = 'typeprefix' or
        strLower = 'union' or
        strLower = 'unsigned' or
        strLower = 'uses' or
        strLower = 'valuebase' or
        strLower = 'valuetype' or
        strLower = 'void' or
        strLower = 'wchar' or
        strLower = 'wstring'

endpackage
```

B.3 Constraints for *face* Package

```
package face

  context Element
    /*
     * Check that name is a valid identifier.
     */
    inv nameIsValidIdentifier:
      not (self.oclIsTypeOf(face::conceptual::Generalization) or
           self.oclIsTypeOf(face::logical::Generalization) or
           self.oclIsTypeOf(face::platform::Generalization) or
           self.oclIsTypeOf(face::logical::Constraint))
      implies
        Element::isValidIdentifier(self.name)

  context ConceptualDataModel
    /*
     * Ensure that Generalization is in the same container as the specialized Entity.
     */
    inv conceptualGeneralizationSameContainerAsSpecialized:
      self.element
        ->select(e | e.oclIsTypeOf(face::conceptual::Generalization))
        ->collect(g | g.oclAsType(face::conceptual::Generalization))
        ->forAll(g | self.element->exists(e | e = g.specialized))

  context LogicalDataModel
    /*
     * Ensure that Generalization is in the same container as the specialized Entity.
     */
    inv logicalGeneralizationSameContainerAsSpecialized:
      self.element
        ->select(e | e.oclIsTypeOf(face::logical::Generalization))
        ->collect(g | g.oclAsType(face::logical::Generalization))
        ->forAll(g | self.element->exists(e | e = g.specialized))

  context PlatformDataModel
    /*
     * Ensure that Generalization is in the same container as the specialized Entity.
     */
    inv platformGeneralizationSameContainerAsSpecialized:
```

```
      self.element
        ->select(e | e.oclIsTypeOf(face::platform::Generalization))
        ->collect(g | g.oclAsType(face::platform::Generalization))
        ->forAll(g | self.element->exists(e | e = g.specialized))

endpackage
```

B.4 Constraints for *face::conceptual* Package

```
package face

  context Element
    /*
     * Get conceptual association by name.
     */
    static def: getConceptualAssociationByName(name : String)
     : face::conceptual::Association =
      let allAssociations : Collection(face::conceptual::Association) =
        face::conceptual::Association.allInstances() in
      -- there should be exactly one Association with expected name
      if allAssociations->one(a | a.name = name)
      then
        allAssociations->any(a | a.name = name)
      else
        null
      endif

    /*
     * Get composition path resolution.
     */
    static def: getConceptualAssociatedEntityFromToken( association :
face::conceptual::Association, token : String)
     : face::conceptual::Characteristic =

      let currPathTokenSplit : Sequence(String) =
          Element::tokenizeString(token.replaceAll(']', ''),'[') in
      let tokenRolename : String = currPathTokenSplit->first() in
      let allAssociations : Collection(face::conceptual::Association) =
        face::conceptual::Association.allInstances() in

      -- the Association should have exactly one AssociatedEntity with
      -- expected rolename
      if association.associatedEntity->one(c | c.rolename = tokenRolename)
      then
        association.associatedEntity->any(c | c.rolename = tokenRolename)
      else
        null
      endif

    /*
     * Get composition path resolution.
     */
    static def: getConceptualPathResolution(ce : face::conceptual::ComposableElement,
      token : String) : face::conceptual::Characteristic =

      if ce.oclIsKindOf(face::conceptual::Entity)
      then
        ce.oclAsType(face::conceptual::Entity).getCharacteristicByRolename(token)
      else
        null
      endif
```

```
    /*
     * Helper method to determine if a path relative to a ComposableElement is valid.
     */
    static def: resolveConceptualPath(ce : face::conceptual::ComposableElement,
pathTokens : Sequence(String)) : Sequence(face::conceptual::Characteristic) =
      if pathTokens->size() = 0
      then
        Sequence{}
      else
        let token : String = pathTokens->first() in
        if token.indexOf('[') > 0
        then
          let tokenSplit : Sequence(String) =
            Element::tokenizeString(token.replaceAll(']', ''),'[') in
          let rolename : String = tokenSplit->first() in
          let associationName : String = tokenSplit->last() in
          let association : face::conceptual::Association =
Element::getConceptualAssociationByName(associationName) in
            if association <> null
            then
              let resolvedCharacteristic : face::conceptual::Characteristic =
association.getCharacteristicByRolename(rolename) in
              if resolvedCharacteristic = null
              then
                Sequence{null}
              else
                Element::resolveConceptualPath(association,
Element::removeFirstString(pathTokens))
                  ->prepend(resolvedCharacteristic)
              endif
            else
              Sequence{null}
            endif
        else
          let resolvedCharacteristic : face::conceptual::Characteristic =
Element::getConceptualPathResolution(ce, token) in
          if resolvedCharacteristic = null
          then
            Sequence{null}
          else
            Element::resolveConceptualPath(resolvedCharacteristic.getType(),
Element::removeFirstString(pathTokens))
              ->prepend(resolvedCharacteristic)
          endif
        endif
      endif

endpackage

package face::conceptual

  context Element
    /*
     * Every face::conceptual::Element in the FACE data model shall
     * have a unique name.
     */
    inv hasUniqueName:
      let otherConceptualElements: Set(face::conceptual::Element) =
        face::conceptual::Element.allInstances()
          ->excluding(self)
          ->select(e | not e.oclIsTypeOf(face::conceptual::Generalization))
      in
```

```
      self.oclIsTypeOf(face::conceptual::Generalization) or
      not otherConceptualElements.name->exists(e | e = self.name)

  context Entity
    /*
     * Get characteristic by rolename.
     */
    def: getCharacteristicByRolename(rolename : String)
     : face::conceptual::Characteristic =
      let characteristics : Set(face::conceptual::Characteristic) =
self.getCharacteristics() in
      if characteristics->one(c | c.rolename = rolename)
      then
        characteristics->any(c | c.rolename = rolename)
      else
        null
      endif

    /*
     * A helper method that returns the Identity of an Entity.
     */
    def: getEntityIdentity() : Bag(OclAny) =
      self.getCharacteristics()->collect(c | c.getIdentityContribution())->asBag()

    /*
     * A helper method that returns the Characteristics of an Entity.
     */
    def: getCharacteristics() : OrderedSet(face::conceptual::Characteristic) =
      if self.oclIsTypeOf(face::conceptual::Association)
      then
        self.oclAsType(face::conceptual::Association)
          ->collect(associatedEntity.oclAsType(face::conceptual::Characteristic))
          ->union(self.composition)->asOrderedSet()
      else
        self.composition
      endif

    /*
     * The full composition hierarchy of each conceptual Entity shall be
     * unique. Uniqueness is determined by number, multiplicity and
     * type of its composed ComposableElement.
     */
    inv entityIsUnique:
      let ceIdentity: Bag(OclAny) =
        self.getEntityIdentity() in
        face::conceptual::Entity.allInstances()->excluding(self)
          ->forAll(ce | ce.getEntityIdentity() <> ceIdentity)

    /*
     * Every face::conceptual::Entity in the FACE data model shall contain
     * a face::conceptual::InformationElement named 'UniqueIDType'.
     */
    inv hasUniqueID:
      self.composition.type
      ->exists(a | a.oclIsTypeOf(Observable)
        and a.name = 'UniqueIdentifier'
      )

    /*
     * Every face::conceptual::Characteristic within the scope
     * of an Entity must have a unique name.
     */
    inv allCharacteristicsHaveUniqueRolename:
```

```
      self.getCharacteristics()->isUnique(rolename)

    /*
     * A conceptual Entity must be composed from conceptual
     * BasisElements or other conceptual Entities which
     * are composed of conceptual BasisElements.
     * This constraint ensures an Entity does not include
     * itself within its composition hierarchy.
     */
    inv entityConstructed:
      let ceClosure = self->closure(ce |
        let types = ce.composition.type->asSet()
          in
        let entityTypes = types
          ->select(t | t.oclIsTypeOf(face::conceptual::Entity)) in
        entityTypes->collect(t | t.oclAsType(face::conceptual::Entity))
      ) in
      not ceClosure->includes(self)

  context Generalization

    /*
     * A helper method that returns a bag of candidate identity contributions, one
     * candidate for each specializations of the input type.
    */
    def: candidateIdentityContributions(gc : Sequence(OclAny)) : Bag(OclAny) =
      let specializedType : face::conceptual::ComposableElement =
        gc->first().oclAsType(face::conceptual::ComposableElement) in
      let applicableGeneralizations : Set(face::conceptual::Generalization) =
        face::conceptual::Generalization.allInstances()
          ->select(gen | gen.specialized = specializedType) in
      let candidateReplacementIdentityContributions : Bag(Sequence(OclAny)) =
        applicableGeneralizations.generalized->collectNested(s | Sequence{s, gc-
>at(2), gc->at(3)}) in
      candidateReplacementIdentityContributions

    /*
     * Ensure that the specialized entity has a characteristic that corresponds to
     * each characteristic in the generalized entity. The corresponding specialized
     * characteristic may be of the same type or be a specialized type of the type
     * of the generalized characteristic.
     */
    inv generalizationStatementCorrect:
      let generalizedContents : Sequence(Sequence(OclAny)) =
        self.generalized.getCharacteristics()->collectNested(c |
c.getIdentityContribution()) in

      let specializedContents : Sequence(Sequence(OclAny)) =
        self.specialized.getCharacteristics()->collectNested(c |
c.getIdentityContribution()) in

      let generalizedContentCandidates : Bag(Sequence(OclAny)) =
        specializedContents->collectNested(gc : Sequence(OclAny) |
candidateIdentityContributions(gc))->iterate(
          replacementBag: Bag(Sequence(OclAny));
          acc : Bag(Sequence(OclAny)) = Bag{}|
          acc->union(replacementBag)
        )
      in

      generalizedContents->forAll(gc : Sequence(OclAny) | (
        specializedContents->exists(sc : Sequence(OclAny) | sc = gc) or
        generalizedContentCandidates->exists(scc : Sequence(OclAny) | scc = gc)
```

```
      ))

  context View

    /*
     * A helper method that returns the Identity of an Entity.
     */
    def: getViewIdentity() : Bag(OclAny) =
      self.characteristic->collect(c | c.getIdentityContribution())

    /*
     * The each View shall be unique.  Uniqueness is determined by
     * number, multiplicity, type, and context of its ProjectedCharacteristics.
     */
    inv viewIsUnique:
      let cvIdentity: Bag(OclAny) =
        self.getViewIdentity() in
        face::conceptual::View.allInstances()->excluding(self)
          ->forAll(cv | cv.getViewIdentity() <> cvIdentity
          )

    /*
     * Ensure that the rolename for each characteristic projection is unique
     * within a view. The rolename may be implicit or explicit.
     */
    inv rolenameIsUnique:
      let rolenames : Set(String) = self.characteristic->collect(c | c.getRolename())-
>asSet() in
      rolenames->forAll(rn | rn <> null) and rolenames->isUnique(rn | rn)

  context Characteristic

    def: getType()
     : face::conceptual::ComposableElement =
      if self.oclIsTypeOf(face::conceptual::Composition)
      then
        self.oclAsType(face::conceptual::Composition).type.oclAsType(face::
          conceptual::ComposableElement)
      else
        self.oclAsType(face::conceptual::AssociatedEntity).type.oclAsType(face::
          conceptual::ComposableElement)
      endif

    /*
     * A helper method that returns the contribution that
     * a Characteristic makes to an Entity's identity.
     */
    def: getIdentityContribution() : Sequence(OclAny) =
      Sequence{self.getType(), self.upperBound, self.lowerBound}

    /*
     * For conceptual, logical, and platform Compositions the lowerBound
     * shall be less than or equal to upperBound.
     */
    inv lowerBound_LTE_UpperBound:
      self.upperBound <> -1 implies self.lowerBound <= self.upperBound
    /*
     * For conceptual, logical, and platform Compositions, upperBound shall
     * be == -1 or >= 1.
     */
    inv upperBoundValid:
      self.upperBound = -1 or self.upperBound >= 1
    /*
```

```
    * For conceptual, logical, and platform Compositions, lowerBound shall
    * be >= zero.
    */
  inv lowerBoundValid:
    self.lowerBound  >= 0

  /*
   * Check that rolename is a valid identifier.
   */
  inv rolenameIsValidIdentifier:
    Element::isValidIdentifier(self.rolename)

context AssociatedEntity
  /*
   * AssociatedEntity must have a valid path.
   */
  inv associatedEntityPathValid:
    let ce : face::conceptual::ComposableElement =
      self.type.oclAsType(face::conceptual::ComposableElement) in
    let tokens : Sequence(String) =
      Element::tokenizeString(self.path.replaceAll('->', '.'), '.') in

    Element::isValidPathFormat(self.path) and
    Element::resolveConceptualPath(ce, tokens)->forAll(c | c <> null)

context CharacteristicProjection

  /*
   * CharacteristicProjection must have a valid path.
   */
  inv characteristicProjectionPathValid:
    let ce : face::conceptual::ComposableElement =
      self.projectedCharacteristic.oclAsType(face::conceptual::ComposableElement) in
    let tokens : Sequence(String) =
      Element::tokenizeString(self.path.replaceAll('->', '.'), '.') in

    Element::isValidPathFormat(self.path) and
    Element::resolveConceptualPath(ce, tokens)->forAll(c | c <> null)

  /*
   * A helper method that returns the contribution that
   * a Characteristic makes to an Entity's identity.
   */
  def: getIdentityContribution() : Sequence(OclAny) =
    Sequence{self.projectedCharacteristic, self.path}

  /*
   * A helper method that returns the computed rolename of a projection.
   */
  def: getRolename() : String =
    if self.rolename.size() > 0
    then
      self.rolename
    else
    if self.path.size() > 0 and
      self.path.substring(self.path.size(),self.path.size()) <> ']'
    then
      let ce : face::conceptual::ComposableElement =
        self.projectedCharacteristic.oclAsType(face::conceptual::ComposableElement)
        in
      let tokens : Sequence(String) =
        Element::tokenizeString(self.path.replaceAll('->', '.'), '.') in
```

```
        Element::resolveConceptualPath(ce, tokens)->last().rolename
      else
        null
      endif
      endif

    /*
     * If defined, check that rolename is a valid identifier.
     */
    inv rolenameIsValidIdentifier:
      self.rolename.size() > 0 implies
        Element::isValidIdentifier(self.rolename)

endpackage
```

B.5 Constraints for *face::logical* Package

```
package face

  context Element
    /*
     * Get logical association by name.
     */
    static def: getLogicalAssociationByName(name : String)
     : face::logical::Association =
      let allAssociations : Collection(face::logical::Association) =
        face::logical::Association.allInstances() in
      -- there should be exactly one Association with expected name
      if allAssociations->one(a | a.name = name)
      then
        allAssociations->any(a | a.name = name)
      else
        null
      endif

    /*
     * Get composition path resolution.
     */
    static def: getLogicalAssociatedEntityFromToken( association :
face::logical::Association, token : String)
     : face::logical::Characteristic =

      let currPathTokenSplit : Sequence(String) =
          Element::tokenizeString(token.replaceAll(']', ''),'[') in
      let tokenRolename : String = currPathTokenSplit->first() in
      let allAssociations : Collection(face::logical::Association) =
        face::logical::Association.allInstances() in

      -- the Association should have exactly one AssociatedEntity with expected
      -- rolename
      if association.associatedEntity->one(c | c.rolename = tokenRolename)
      then
        association.associatedEntity->any(c | c.rolename = tokenRolename)
      else
        null
      endif

    /*
     * Get composition path resolution.
     */
```

```
    static def: getLogicalPathResolution(ce : face::logical::ComposableElement, token
: String)
     : face::logical::Characteristic =

      if ce.oclIsKindOf(face::logical::Entity)
      then
        ce.oclAsType(face::logical::Entity).getCharacteristicByRolename(token)
      else
        null
      endif

    /*
     * Helper method to determine if a path relative to a ComposableElement is valid.
     */
    static def: resolveLogicalPath(ce : face::logical::ComposableElement, pathTokens :
Sequence(String)) : Sequence(face::logical::Characteristic) =
      if pathTokens->size() = 0
      then
        Sequence{}
      else
        let token : String = pathTokens->first() in
        if token.indexOf('[') > 0
        then
          let tokenSplit : Sequence(String) =
            Element::tokenizeString(token.replaceAll(']', ''),'[') in
          let rolename : String = tokenSplit->first() in
          let associationName : String = tokenSplit->last() in
          let association : face::logical::Association =
Element::getLogicalAssociationByName(associationName) in
            if association <> null
            then
              let resolvedCharacteristic : face::logical::Characteristic =
association.getCharacteristicByRolename(rolename) in
              if resolvedCharacteristic = null
              then
                Sequence{null}
              else
                Element::resolveLogicalPath(association,
Element::removeFirstString(pathTokens))
                  ->prepend(resolvedCharacteristic)
              endif
            else
              Sequence{null}
            endif
        else
          let resolvedCharacteristic : face::logical::Characteristic =
Element::getLogicalPathResolution(ce, token) in
          if resolvedCharacteristic = null
          then
            Sequence{null}
          else
            Element::resolveLogicalPath(resolvedCharacteristic.getType(),
Element::removeFirstString(pathTokens))
              ->prepend(resolvedCharacteristic)
          endif
        endif
      endif

endpackage

package face::logical

  context Element
```

```
    /*
     * Every face::logical::Element except Constraint and Generalization shall
     * have a unique name.
     */
    inv hasUniqueName:
      let otherLogicalElements: Set(face::logical::Element) =
        face::logical::Element.allInstances()
          ->excluding(self)
          ->select(e | not e.oclIsTypeOf(face::logical::Generalization))
          ->select(e | not e.oclIsTypeOf(face::logical::Constraint))
      in
      self.oclIsTypeOf(face::logical::Generalization) or
      self.oclIsTypeOf(face::logical::Constraint) or
      not otherLogicalElements.name->exists(e | e = self.name)

  context Entity
    /*
     * Get characteristic by rolename.
     */
    def: getCharacteristicByRolename(rolename : String)
     : face::logical::Characteristic =
      let characteristics : Set(face::logical::Characteristic) =
self.getCharacteristics() in
      if characteristics->one(c | c.rolename = rolename)
      then
        characteristics->any(c | c.rolename = rolename)
      else
        null
      endif

    /*
     * A helper method that returns the Characteristics of an Entity.
     */
    def: getCharacteristics() : OrderedSet(face::logical::Characteristic) =
      if self.oclIsTypeOf(face::logical::Association)
      then
        self.oclAsType(face::logical::Association)
          ->collect(associatedEntity.oclAsType(face::logical::Characteristic))
          ->union(self.composition)->asOrderedSet()
      else
        self.composition
      endif

    /*
     * Every face::logical::Characteristic within the scope
     * of an Entity must have a unique name.
     */
    inv allCharacteristicsHaveUniqueRolename:
      self.getCharacteristics()->isUnique(rolename)

    /*
     * Ensure that the Compositions in a logical Entity realize Compositions in the
     * conceptual Entity that the logical Entity realizes.
     */
    inv logicalEntityConsistentWithConceptual:
      self.composition.realizes->forAll(c | self.realizes.composition->exists(c2 | c =
c2))

  context Generalization

    /*
     * A helper method that returns a bag of candidate identity contributions, one
     * candidate for each specializations of the input type.
```

```
     */
    def: candidateIdentityContributions(gc : Sequence(OclAny)) : Bag(OclAny) =
      let specializedType : face::logical::ComposableElement =
        gc->first().oclAsType(face::logical::ComposableElement) in
      let applicableGeneralizations : Set(face::logical::Generalization) =
        face::logical::Generalization.allInstances()
          ->select(gen | gen.specialized = specializedType) in
      let candidateReplacementIdentityContributions : Bag(Sequence(OclAny)) =
        applicableGeneralizations.generalized->collectNested(s | Sequence{s, gc-
>at(2), gc->at(3)}) in
      candidateReplacementIdentityContributions

    /*
     * Ensure that the specialized entity has a characteristics that corresponds to
     * each characteristic in the generalized entity. The corresponding specialized
     * characteristic may be of the same type or be a specialized type of the type of
     * the generalized characteristic.
    */
    inv generalizationStatementCorrect:
      let generalizedContents : Sequence(Sequence(OclAny)) =
        self.generalized.getCharacteristics()->collectNested(c |
c.getIdentityContribution()) in

      let specializedContents : Sequence(Sequence(OclAny)) =
        self.specialized.getCharacteristics()->collectNested(c |
c.getIdentityContribution()) in

      let generalizedContentCandidates : Bag(Sequence(OclAny)) =
        specializedContents->collectNested(gc : Sequence(OclAny) |
candidateIdentityContributions(gc))->iterate(
          replacementBag: Bag(Sequence(OclAny));
          acc : Bag(Sequence(OclAny)) = Bag{}|
          acc->union(replacementBag)
        )
      in

      generalizedContents->forAll(gc : Sequence(OclAny) | (
        specializedContents->exists(sc : Sequence(OclAny) | sc = gc) or
        generalizedContentCandidates->exists(scc : Sequence(OclAny) | scc = gc)
      ))

  context Association

    /*
     * Ensure that the AssociatedEntities in a logical Association realize
     * AssociatedEntities in the conceptual Association that the logical
     * Association realizes.
     */
    inv logicalAssociationConsistentWithConceptual:
      let conceptualAssociationContents: Bag(face::conceptual::AssociatedEntity) =
        self.realizes.oclAsType(face::conceptual::Association).associatedEntity in
      self.associatedEntity.realizes->forAll(ae | conceptualAssociationContents-
>exists(ae2 | ae = ae2))

  context Characteristic
    /*
     * Helper method to get the type of a concrete Characteristic.
    */
    def: getType() : face::logical::ComposableElement =
      if self.oclIsTypeOf(face::logical::Composition)
      then
        self.oclAsType(face::logical::Composition).type.oclAsType(face::logical::
          ComposableElement)
```

```
    else
      self.oclAsType(face::logical::AssociatedEntity).type.oclAsType(face::
        logical::ComposableElement)
    endif

  /*
   * Helper method to get the realized characteristic of a concrete Characteristic.
   */
  def: getRealizes() : face::conceptual::Characteristic =
    if self.oclIsTypeOf(face::logical::Composition)
    then
      self.oclAsType(face::logical::Composition).realizes.oclAsType(face::
        conceptual::Characteristic)
    else
      self.oclAsType(face::logical::AssociatedEntity).realizes.oclAsType(face::
        conceptual::Characteristic)
    endif

  /*
   * A helper method that returns the contribution that
   * a Characteristic makes to an Entity's identity.
   */
  def: getIdentityContribution() : Sequence(OclAny) =
    Sequence{self.getType(), self.upperBound, self.lowerBound}

  /*
   * Check that rolename is a valid identifier.
   */
  inv rolenameIsValidIdentifier:
    Element::isValidIdentifier(self.rolename)

context Composition
  /*
   * Ensure that when an element realizes another element, the
   * upper and lower bounds of the realized entity match those
   * of the realizing entity.
   */
  inv logicalBoundsEqualConceptual:
    self.lowerBound = self.realizes.lowerBound and
    self.upperBound = self.realizes.upperBound

  /* A logical entity composition hierarchy must be consistent
   * with the composition hierarchy of the conceptual entity
   * that it realizes. The logical measurements must correspond
   * with the conceptual observables.
   */
  inv logicalCompositionConsistentWithConceptual:
    if self.type.oclIsKindOf(face::logical::Entity) then
      self.type.oclAsType(face::logical::Entity).realizes = self.realizes.type
    else
    if self.type.oclIsKindOf(face::logical::Measurement) then
      self.type.oclAsType(face::logical::Measurement).realizes = self.realizes.type
    else
      false
    endif
    endif

context AssociatedEntity

  /*
   * AssociatedEntity must have a valid path.
   */
  inv associatedEntityPathValid:
```

```
    let ce : face::logical::ComposableElement =
      self.type.oclAsType(face::logical::ComposableElement) in
    let tokens : Sequence(String) =
      Element::tokenizeString(self.path.replaceAll('->', '.'), '.') in

    Element::isValidPathFormat(self.path) and
    Element::resolveLogicalPath(ce, tokens)->forAll(c | c <> null)

  /*
   * The type of a logical AssociatedEntity must realize the same conceptual type
   * that is the type of the realized conceptual AssociatedEntity.
   */
  inv logicalAssociatedEntityConsistentWithConceptual:
    self.type.realizes = self.realizes.type

/*
 * Ensure that when an element realizes another element, the
 * upper and lower bounds of the realized entity match those
 * of the realizing entity.
 */
inv logicalBoundsEqualConceptual:
  self.lowerBound = self.realizes.lowerBound and
  self.upperBound = self.realizes.upperBound

context CharacteristicProjection

  /*
   * CharacteristicProjection must have a valid path.
   */
  inv characteristicProjectionPathValid:
    let ce : face::logical::ComposableElement =
      self.projectedCharacteristic.oclAsType(face::logical::ComposableElement) in
    let tokens : Sequence(String) =
      Element::tokenizeString(self.path.replaceAll('->', '.'), '.') in

    Element::isValidPathFormat(self.path) and
    Element::resolveLogicalPath(ce, tokens)->forAll(c | c <> null)

  /*
   * If a logical CharacteristicProjection realizes a conceptual
   * CharacteristicProjection the path must be follow the same path
   * as the conceptual.
   */
  inv logicalCharacteristicProjectionConsistentWithConcepual:
    self.realizes <> null implies

      let ce : face::logical::ComposableElement =
        self.projectedCharacteristic.oclAsType(face::logical::ComposableElement) in
      let tokens : Sequence(String) =
        Element::tokenizeString(self.path.replaceAll('->', '.'), '.') in
      let logicalPath : Sequence(face::logical::Characteristic) =
        Element::resolveLogicalPath(ce, tokens) in
      let conceptualCE : face::conceptual::ComposableElement =
        self.realizes.projectedCharacteristic.oclAsType(face::
          conceptual::ComposableElement) in
      let conceptualTokens : Sequence(String) =
        Element::tokenizeString(self.realizes.path.replaceAll('->', '.'), '.') in
      let conceptualPath : Sequence(face::conceptual::Characteristic) =
        Element::resolveConceptualPath(conceptualCE, conceptualTokens) in

      logicalPath->collect(c | c.getRealizes()) = conceptualPath

  /*
```

```
     * If defined, check that rolename is a valid identifier.
     */
    inv rolenameIsValidIdentifier:
      self.rolename.size() > 0 implies
        Element::isValidIdentifier(self.rolename)

    /*
     * A helper method that returns the computed rolename of a projection.
     */
    def: getRolename() : String =
      if self.rolename.size() > 0
      then
        self.rolename
      else
      if self.path.size() > 0 and
        self.path.substring(self.path.size(),self.path.size()) <> ']'
      then
        let ce : face::logical::ComposableElement =
          self.projectedCharacteristic.oclAsType(face::logical::ComposableElement) in
        let tokens : Sequence(String) =
          Element::tokenizeString(self.path.replaceAll('->', '.'), '.') in

        Element::resolveLogicalPath(ce, tokens)->last().rolename
      else
        null
      endif
      endif

  context View
    /*
     * If a logical View realizes a conceptual View then all the
     * CharacteristicProjections should realize a conceptual
     * CharacteristicProjectin in the conceptual View.
     */
    inv logicalViewConsistentWithConceptual:
      if self.realizes = null
      then
        self.characteristic->forAll(c | c.realizes = null)
      else
        self.characteristic->forAll(c | self.realizes.characteristic->exists(c2 |
c.realizes = c2))
      endif

    /*
     * Ensure that the rolename for each characteristic projection is unique
     * within a view. The rolename may be implicit or explicit.
     */
    inv rolenameIsUnique:
      let rolenames : Set(String) = self.characteristic->collect(c | c.getRolename())-
>asSet() in
      rolenames->forAll(rn | rn <> null) and rolenames->isUnique(rn | rn)

  context ValueTypeUnit
    /*
     * An EnumeratedConstraint should select only labels from the Enumerated
     * instance it is constraining.
     */
    inv appropriateLabelsForEnumeratedConstraint:
      if self.constraint <> null
      then
        if self.constraint.oclIsTypeOf(face::logical::EnumerationConstraint)
        then
          self.valueType.oclIsTypeOf(face::logical::Enumerated) and
```

```
          self.constraint.oclAsType(face::
            logical::EnumerationConstraint).allowedValue->forAll(av |
            self.valueType.oclAsType(face::logical::Enumerated).label->exists(l | l =
av)
          )
        else
          true -- constraint is not an EnumerationConstraint
        endif
      else
        true -- with no constraint there is nothing to check
      endif

   context ValueType
     /*
     * The name of a ValueType instance should match the metaclass for except
     * for Enumerated ValueTypes.
     */
     inv nameOfValueTypeMatchesNameOfMetaclass:
       self.oclIsTypeOf(face::logical::Enumerated) or self.name = self.oclType().name

   context MeasurementSystem

     def: hasAnEnumeratedValueType() : Boolean =

       let valueTypes: Collection(face::logical::ValueType) =
self.measurementSystemAxis.defaultValueTypeUnit.valueType in

       valueTypes->exists(vt | vt.oclIsTypeOf(face::logical::Enumerated))

     /*
      * Only one measurement system with an Enumerated ValueType.
      */
     inv onlyOneEnumeratedMeasurementSystem:
       if self.name = 'AbstractDiscreteSet'
       then
         self.hasAnEnumeratedValueType() and
self.measurementSystemAxis.defaultValueTypeUnit->size() = 1
       else
         not self.hasAnEnumeratedValueType()
       endif

     /*
      * Ensure MeasurementSystemAxes have CoordinateSystemAxes that
      * are in the CoordinateSystem of the MeasurementSystem
      */
     inv measurementSystemConsistentWithCoordinateSystem:
       self.measurementSystemAxis.axis->asSet() = coordinateSystem.axis->asSet()

     /*
      * Ensure that ReferencePoint uses MeasurementSystemAxes and ValueTypeUnits
      * from the correct MeasurementSystem.
      */
     inv referencePointPartConsistentWithAxes:
       self.referencePoint.referencePointPart->forAll(rpp |
         rpp.axis->forAll(
           rppAxis | self.measurementSystemAxis->exists(msa | msa = rppAxis)
         ) and
         rpp.valueTypeUnit->forAll(
           rppVTU | self.measurementSystemAxis.defaultValueTypeUnit->exists(vtu | vtu
= rppVTU)
         )
       )
```

```
   context Measurement
     /*
      * Helper method to check if a Measurement has an Enumerated ValueType.
      */
     def: hasAnEnumeratedValueType() : Boolean =
       let valueTypes: Collection(face::logical::ValueType) =
             self.measurementAxis.valueTypeUnit.valueType in
       valueTypes->exists(vt | vt.oclIsTypeOf(face::logical::Enumerated))

     /*
      * Ensure that all Measurements with an Enumerated ValueType are associated
      * with the single 'AbstractDiscreteSet'.
      */
     inv allEnumeratedMeasurementUseEnumeratedMeasurementSystem:
       if self.hasAnEnumeratedValueType()
       then
         self.measurementSystem.name = 'AbstractDiscreteSet'
       else
         self.measurementSystem.name <> 'AbstractDiscreteSet'
       endif

     /*
      * Ensure MeasurementAxes have MeasurementSystemAxes that are in the
      * MeasurementSystem of the Measurement.
      */
     inv measurementConsistentWithMeasurementSystem:
       self.measurementAxis.measurementSystemAxis->asSet() =
measurementSystem.measurementSystemAxis->asSet()

endpackage
```

B.6 Constraints for *face::platform* Package

```
package face

  context Element
    /*
     * Get platform association by name.
     */
    static def: getPlatformAssociationByName(name : String)
     : face::platform::Association =
      let allAssociations : Collection(face::platform::Association) =
        face::platform::Association.allInstances() in
      -- there should be exactly one Association with expected name
      if allAssociations->one(a | a.name = name)
      then
        allAssociations->any(a | a.name = name)
      else
        null
      endif

    /*
     * Get composition path resolution.
     */
    static def: getPlatformAssociatedEntityFromToken( association :
face::platform::Association, token : String)
     : face::platform::Characteristic =

      let currPathTokenSplit : Sequence(String) =
          Element::tokenizeString(token.replaceAll(']', ''),'[') in
      let tokenRolename : String = currPathTokenSplit->first() in
```

```
      let allAssociations : Collection(face::platform::Association) =
        face::platform::Association.allInstances() in

      -- the Association should have exactly one AssociatedEntity with
      -- expected rolename
      if association.associatedEntity->one(c | c.rolename = tokenRolename)
      then
        association.associatedEntity->any(c | c.rolename = tokenRolename)
      else
        null
      endif

    /*
     * Get composition path resolution
     */
    static def: getPlatformPathResolution(ce : face::platform::ComposableElement,
token : String)
     : face::platform::Characteristic =

      if ce.oclIsKindOf(face::platform::Entity)
      then
        ce.oclAsType(face::platform::Entity).getCharacteristicByRolename(token)
      else
        null
      endif

    /*
     * Helper method to determine if a path relative to a ComposableElement is valid.
     */
    static def: resolvePlatformPath(ce : face::platform::ComposableElement, pathTokens
: Sequence(String)) : Sequence(face::platform::Characteristic) =
      if pathTokens->size() = 0
      then
        Sequence{}
      else
        let token : String = pathTokens->first() in
        if token.indexOf('[') > 0
        then
          let tokenSplit : Sequence(String) =
            Element::tokenizeString(token.replaceAll(']', ''),'[') in
          let rolename : String = tokenSplit->first() in
          let associationName : String = tokenSplit->last() in
          let association : face::platform::Association =
Element::getPlatformAssociationByName(associationName) in
            if association <> null
            then
              let resolvedCharacteristic : face::platform::Characteristic =
association.getCharacteristicByRolename(rolename) in
              if resolvedCharacteristic = null
              then
                Sequence{null}
              else
                Element::resolvePlatformPath(association,
Element::removeFirstString(pathTokens))
                  ->prepend(resolvedCharacteristic)
              endif
            else
              Sequence{null}
            endif
        else
          let resolvedCharacteristic : face::platform::Characteristic =
Element::getPlatformPathResolution(ce, token) in
          if resolvedCharacteristic = null
```

```
          then
            Sequence{null}
          else
            Element::resolvePlatformPath(resolvedCharacteristic.getType(),
Element::removeFirstString(pathTokens))
              ->prepend(resolvedCharacteristic)
          endif
        endif
      endif

endpackage

package face::platform

  context Element
    /*
     * Every face::platform::Element except Generalization shall
     * have a unique name.
     */
    inv hasUniqueName:
      let otherPlatformElements: Set(face::platform::Element) =
        face::platform::Element.allInstances()
          ->excluding(self)
          ->select(e | not e.oclIsTypeOf(face::platform::Generalization))
      in
      self.oclIsTypeOf(face::platform::Generalization) or
      not otherPlatformElements.name->exists(e | e = self.name)

  context Composition
    /*
     * Ensure that when an element realizes another element, the
     * upper and lower bounds of the realized entity match those
     * of the realizing entity.
     */
    inv platformBoundsEqualLogical:
      self.lowerBound = self.realizes.lowerBound and
      self.upperBound = self.realizes.upperBound

    /* A platform entity composition hierarchy must be consistent
     * with the composition hierarchy of the logical entity
     * that it realizes. The platform value types must correspond
     * with the logical measurements and information elements.
     */
    inv platformCompositionConsistentWithLogical:
      if self.type.oclIsKindOf(face::platform::Entity) then
        self.type.oclAsType(face::platform::Entity).realizes = self.realizes.type
      else
      if self.type.oclIsKindOf(face::platform::IDLType) then
        self.type.oclAsType(face::platform::IDLType).realizes = self.realizes.type
      else
        false
      endif
      endif

  context Characteristic
    /*
     * Helper method to get the type of a concrete Characteristic.
     */
    def: getType()
     : face::platform::ComposableElement =
      if self.oclIsTypeOf(face::platform::Composition)
      then
        self.oclAsType(face::platform::Composition).type.oclAsType(face::
```

```
          platform::ComposableElement)
      else
        self.oclAsType(face::platform::AssociatedEntity).type.oclAsType(face::
          platform::ComposableElement)
      endif

    /*
     * Helper method to get the realized characteristic of a concrete Characteristic.
     */
    def: getRealizes() : face::logical::Characteristic =
      if self.oclIsTypeOf(face::platform::Composition)
      then
        self.oclAsType(face::platform::Composition).realizes.oclAsType(face::
          logical::Characteristic)
      else
        self.oclAsType(face::platform::AssociatedEntity).realizes.oclAsType(face::
          logical::Characteristic)
      endif

    /*
     * A helper method that returns the contribution that
     * a Characteristic makes to an Entity's identity.
     */
    def: getIdentityContribution() : Sequence(OclAny) =
      Sequence{self.getType(), self.upperBound, self.lowerBound}

    /*
     * Check that rolename is a valid identifier.
     */
    inv rolenameIsValidIdentifier:
      Element::isValidIdentifier(self.rolename)

  context Entity
    /*
     * Get characteristic by rolename.
     */
    def: getCharacteristicByRolename(rolename : String)
     : face::platform::Characteristic =
      let characteristics : Set(face::platform::Characteristic) =
self.getCharacteristics() in
      if characteristics->one(c | c.rolename = rolename)
      then
        characteristics->any(c | c.rolename = rolename)
      else
        null
      endif

    /*
     * A helper method that returns the Characteristics of an Entity.
     */
    def: getCharacteristics() : OrderedSet(face::platform::Characteristic) =
      if self.oclIsTypeOf(face::platform::Association)
      then
        self.oclAsType(face::platform::Association)
          ->collect(associatedEntity.oclAsType(face::platform::Characteristic))
          ->union(self.composition)->asOrderedSet()
      else
        self.composition
      endif

    /*
     * Every face::logical::Characteristic within the scope
     * of an Entity must have a unique name.
```

```
     */
    inv allCharacteristicsHaveUniqueRolename:
      self.getCharacteristics()->isUnique(rolename)

    /*
     * Ensure that the Compositions in a platform Entity realize Compositions in the
     * logical Entity that the platform Entity realizes.
     */
    inv platformEntityConsistentWithLogical:
      self.composition.realizes->forAll(c | self.realizes.composition->exists(c2 | c =
c2))

  context Association

    /*
     * Ensure that the AssociatedEntities in a platform Association realize
     * AssociatedEntities in the logical Association that the platform
     * Association realizes.
     */
    inv platformAssociationConsistentWithLogical:
      let logicalAssociationContents: Bag(face::logical::AssociatedEntity) =
        self.realizes.oclAsType(face::logical::Association).associatedEntity in
      self.associatedEntity.realizes->forAll(ae | logicalAssociationContents-
>exists(ae2 | ae = ae2))

  context Generalization

    /*
     * A helper method that returns a bag of candidate identity contributions, one
     * candidate for each specializations of the input type.
     */
    def: candidateIdentityContributions(gc : Sequence(OclAny)) : Bag(OclAny) =
      let specializedType : face::platform::ComposableElement =
        gc->first().oclAsType(face::platform::ComposableElement) in
      let applicableGeneralizations : Set(face::platform::Generalization) =
        face::platform::Generalization.allInstances()
          ->select(gen | gen.specialized = specializedType) in
      let candidateReplacementIdentityContributions : Bag(Sequence(OclAny)) =
        applicableGeneralizations.generalized->collectNested(s | Sequence{s, gc-
>at(2), gc->at(3)}) in
      candidateReplacementIdentityContributions

    /*
     * Ensure that the specialized entity has a characteristics that corresponds to
     * each characteristic in the generalized entity. The corresponding specialized
     * characteristic may be of the same type or be a specialized type of the
     * type of the generalized characteristic.
     */
    inv generalizationStatementCorrect:
      let generalizedContents : Sequence(Sequence(OclAny)) =
        self.generalized.getCharacteristics()->collectNested(c |
c.getIdentityContribution()) in

      let specializedContents : Sequence(Sequence(OclAny)) =
        self.specialized.getCharacteristics()->collectNested(c |
c.getIdentityContribution()) in

      let generalizedContentCandidates : Bag(Sequence(OclAny)) =
        specializedContents->collectNested(gc : Sequence(OclAny) |
candidateIdentityContributions(gc))->iterate(
          replacementBag: Bag(Sequence(OclAny));
          acc : Bag(Sequence(OclAny)) = Bag{}|
          acc->union(replacementBag)
```

```
        )
      in

      generalizedContents->forAll(gc : Sequence(OclAny) | (
        specializedContents->exists(sc : Sequence(OclAny) | sc = gc) or
        generalizedContentCandidates->exists(scc : Sequence(OclAny) | scc = gc)
      ))

  context AssociatedEntity
    /*
     * AssociatedEntity must have a valid path.
     */
    inv associatedEntityPathValid:
      let ce : face::platform::ComposableElement =
        self.type.oclAsType(face::platform::ComposableElement) in
      let tokens : Sequence(String) =
        Element::tokenizeString(self.path.replaceAll('->', '.'), '.') in

      Element::isValidPathFormat(self.path) and
      Element::resolvePlatformPath(ce, tokens)->forAll(c | c <> null)

    /*
     * The type of a platform AssociatedEntity must realize the same logical type
     * that is the type of the realized logical AssociatedEntity.
     */
    inv platformAssociatedEntityConsistentWithLogical:
      self.type.realizes = self.realizes.type

    /*
     * Ensure that when an element realizes another element, the
     * upper and lower bounds of the realized entity match those
     * of the realizing entity.
    */
    inv platformBoundsEqualLogical:
      self.lowerBound = self.realizes.lowerBound and
      self.upperBound = self.realizes.upperBound

  context View

    /*
     * If a platform View realizes a logical View then all the
     * CharacteristicProjections should realize a logical
     * CharacteristicProjectin in the logical View.
     */
    inv platformViewConsistentWithLogical:
      if self.realizes = null
      then
        self.characteristic->forAll(c | c.realizes = null)
      else
        self.characteristic->forAll(c | self.realizes.characteristic->exists(c2 |
c.realizes = c2))
      endif

    /*
     * Ensure that the rolename for each characteristic projection is unique
     * within a view. The rolename may be implicit or explicit.
     */
    inv rolenameIsUnique:
      let rolenames : Set(String) = self.characteristic->collect(c | c.getRolename())-
>asSet() in
      rolenames->forAll(rn | rn <> null) and rolenames->isUnique(rn | rn)

  context CharacteristicProjection
```

```
/*
 * CharacteristicProjection must have a valid path.
 */
inv characteristicProjectionPathValid:
  let ce : face::platform::ComposableElement =
    self.projectedCharacteristic.oclAsType(face::platform::ComposableElement) in
  let tokens : Sequence(String) =
    Element::tokenizeString(self.path.replaceAll('->', '.'), '.') in

  Element::isValidPathFormat(self.path) and
  Element::resolvePlatformPath(ce, tokens)->forAll(c | c <> null)

/*
 * If a platform CharacteristicProjection realizes a logical
 * CharacteristicProjection the path must be follow the same path as the logical.
 */
inv platformCharacteristicProjectionConsistentWithLogical:
  self.realizes <> null implies

    let ce : face::platform::ComposableElement =
      self.projectedCharacteristic.oclAsType(face::platform::ComposableElement) in
    let tokens : Sequence(String) =
      Element::tokenizeString(self.path.replaceAll('->', '.'), '.') in
    let platformPath : Sequence(face::platform::Characteristic) =
      Element::resolvePlatformPath(ce, tokens) in
    let logicalCE : face::logical::ComposableElement =
      self.realizes.projectedCharacteristic.oclAsType(face::
        logical::ComposableElement) in
    let logicalTokens : Sequence(String) =
      Element::tokenizeString(self.realizes.path.replaceAll('->', '.'), '.') in
    let logicalPath : Sequence(face::logical::Characteristic) =
      Element::resolveLogicalPath(logicalCE, logicalTokens) in

    platformPath->collect(c | c.getRealizes()) = logicalPath

/*
 * If defined, check that rolename is a valid identifier.
 */
inv rolenameIsValidIdentifier:
  self.rolename.size() > 0 implies
    Element::isValidIdentifier(self.rolename)

/*
 * A helper method that returns the computed rolename of a projection.
 */
def: getRolename() : String =
  if self.rolename.size() > 0
  then
    self.rolename
  else
  if self.path.size() > 0 and
    self.path.substring(self.path.size(),self.path.size()) <> ']'
  then
    let ce : face::platform::ComposableElement =
      self.projectedCharacteristic.oclAsType(face::platform::ComposableElement) in
    let tokens : Sequence(String) =
      Element::tokenizeString(self.path.replaceAll('->', '.'), '.') in

    Element::resolvePlatformPath(ce, tokens)->last().rolename
  else
    null
  endif
```

```
      endif

  context IDLType

    /*
     * A helper method that returns the ValueTypeUnit collection for a
     * MeasurementAxis.
     */
    static def: getValueUnitTypes(measurementAxis : face::logical::MeasurementAxis)
     : OrderedSet(face::logical::ValueTypeUnit) =

      let measurementAxisContents: OrderedSet(face::logical::ValueTypeUnit) =
measurementAxis.valueTypeUnit in
      if measurementAxisContents->size() > 0
      then
        -- return the overridden ValueTypeUnit collection of the MeasurementAxis
        measurementAxisContents
      else
        -- return the default ValueTypeUnit collection since there is no override
        let measurementSystemAxis: face::logical::MeasurementSystemAxis =
          measurementAxis.measurementSystemAxis in
        measurementSystemAxis.defaultValueTypeUnit
      endif

  context IDLStruct
    /*
     *  An IDL struct cannot realize a logical value type unit.
     */
    inv idlStructDoesNotRealizeValueTypeUnit:
      not self.realizes.oclIsTypeOf(face::logical::ValueTypeUnit)

    inv idlStructRealizesMultiPartMeasurement:
      -- check that realization is to a multi-part measurement
      let abstractMeasurement: face::logical::AbstractMeasurement = self.realizes in
      if abstractMeasurement.oclIsTypeOf(face::logical::Measurement)
      then
        let measurement : face::logical::Measurement =
          abstractMeasurement.oclAsType(face::logical::Measurement) in
        let measurementAxes : Sequence(face::logical::MeasurementAxis) =
measurement.measurementAxis in
        if measurementAxes->size() > 1
        then
          -- the IDL struct realizes a measurement with more than one axis
          true
        else
          -- if the measurement has a single axis ensure that the axis is multi part
          let measurementAxisContents: Sequence(face::logical::ValueTypeUnit) =
            IDLType::getValueUnitTypes(measurementAxes->first()) in
          measurementAxisContents->size() > 1
        endif
      else
      if abstractMeasurement.oclIsTypeOf(face::logical::MeasurementAxis)
      then
        -- ensure that the axis is multi part
        let measurementAxisContents: Sequence(face::logical::ValueTypeUnit) =
          IDLType::getValueUnitTypes(abstractMeasurement.oclAsType(face::
            logical::MeasurementAxis)) in
        measurementAxisContents->size() > 1
      else
        -- abstractMeasurement is a ValueTypeUnit which is atomic
        false
      endif
      endif
```

```
    /*
     * Ensure that the contents of an IDL struct realize the contents of the
     * measurement or measurement axis which the struct realizes.
     */
    inv idlStructConsistentWithLogical:
      -- check that struct is consistent with the logical model
      let containedTypeRealizations: Collection(face::logical::AbstractMeasurement) =
        self.composition.type.realizes in
      let abstractMeasurementContainedParts:
OrderedSet(face::logical::AbstractMeasurement) =

        if self.realizes.oclIsTypeOf(face::logical::Measurement)
        then
          let measurement : face::logical::Measurement =
            self.realizes.oclAsType(face::logical::Measurement) in
          let measurementAxes : Sequence(face::logical::MeasurementAxis) =
measurement.measurementAxis in
          if measurementAxes->size() > 1
          then
            measurementAxes->collect(c |
c.oclAsType(face::logical::AbstractMeasurement))->asSet()
          else
            IDLType::getValueUnitTypes(measurementAxes-
>any(true).oclAsType(face::logical::MeasurementAxis))
          endif
        else
        if self.realizes.oclIsTypeOf(face::logical::MeasurementAxis)
        then
          IDLType::getValueUnitTypes(self.realizes.oclAsType(face::
            logical::MeasurementAxis))
        else
          -- ValueTypeUnit cannot contain anything
          OrderedSet{}
        endif
        endif
      in

      abstractMeasurementContainedParts->asSet() = containedTypeRealizations->asSet()

  context IDLPrimitive

    /*
     *  An IDL primitive can only realize an abstract
     *  measurement that has a single value type unit.
     */
    inv idlPrimitiveRealizesAtomicAbstractMeasurement:
      -- check that realization is to an atomic measurement
      let abstractMeasurement: face::logical::AbstractMeasurement = self.realizes in
      if abstractMeasurement.oclIsTypeOf(face::logical::Measurement)
      then
        let measurement : face::logical::Measurement =
          abstractMeasurement.oclAsType(face::logical::Measurement) in
        let measurementAxes : Sequence(face::logical::MeasurementAxis) =
measurement.measurementAxis in
        if measurementAxes->size() > 1
        then
          -- the IDL primitive should not realize a measurement with more than one
          -- axis
          false
        else
          -- if the measurement has a single axis ensure that the axis is
          -- not multi part
```

```
        let measurementAxisContents: Sequence(face::logical::ValueTypeUnit) =
          IDLType::getValueUnitTypes(measurementAxes->first()) in
        measurementAxisContents->size() <= 1
      endif
    else
    if abstractMeasurement.oclIsTypeOf(face::logical::MeasurementAxis)
    then
      -- ensure that the axis is not multi part
      let measurementAxisContents: Sequence(face::logical::ValueTypeUnit) =
        IDLType::getValueUnitTypes(abstractMeasurement.oclAsType(face::
          logical::MeasurementAxis)) in
      measurementAxisContents->size() <= 1
    else
      -- abstractMeasurement must be a ValueTypeUnit which is atomic
      true
    endif
    endif

endpackage
```

Acronyms

CCB	Configuration Control Board
CR	Change Request
EMOF	Essential Meta Object Facility
FACE	Future Airborne Capability Environment
ID	Identifier
ITAR	International Traffic in Arms Regulations
OCL	Object Constraint Language
OMG	Object Management Group
SDM	Shared Data Model
TWG	Technical Working Group
UoP	Unit of Portability
USM	Unit of Portability Supplied Model
XMI	XML Metadata Interchange
XML	eXtensible Markup Language