Returning query results as an XML document

This page contains all the procedures necessary for running an X-DEVICE query and returning a result to the user in the form of an XML document.

```
function run_query(LogicProgram) returns XMLDoc
   DeepOrShallow := deep_or_shallow_result(LogicProgram)
   TopElement := compile_xdevice(LogicProgram)
   DocOID := generate_result(TopElement)
   if DeepOrShallow = deep then
      XMLDoc := create_result_schema(TopElement) generate_xml(TopElement,DocOID)
   else
      XMLDoc := create_shallow_schema(TopElement) generate_shallow_xml(TopElement,DocOID)
```

```
function generate_result(TopElement) returns DocNode
   TopElementNodes := ∅
   for_each X in get(X) => TopElement do
      TopElementNodes := TopElementNodes ∪append {X}
   new([DocNode,[uri(['X-Device Server URI/TopElement']),children(TopElementNodes)]])=>xml_doc
```

```
function deep_or_shallow_result(LogicProgram) returns { 'deep' | 'shallow' }
```

This function checks the logic program for a top-level construct. If the `xml_result` (or `xml_sorted`) construct is found then it returns the keyword `deep`, meaning that the result will be a **deep copy** of the tree rooted from the derived class tagged with the `xml_result` construct. If the `shallow_result` construct is found, then the keyword `shallow` is returned, meaning that a **shallow copy** of the derived class will be returned (only attributes of the derived class and not sub-elements!).

```
function generate_xml(TopElement,DocOID) returns DOC
   get_children(TopElements) => DocOID
   DOC := ∅
   for_each X in TopElements do
      SubTree := xml_subtree(TopElement,X)
      DOC := DOC ∪append {SubTree}
```

```
function xml_subtree(Class,Instance) returns Tree
   get_elem_order(ElemOrder) => Class
   get_att_lst(AttList) => Class
   get_alias(Alias) => Class
   get_empty(EmptyAtts) => Class
   Attributes := ∅
   Elements := ∅
   for_each A in AttList do
      get_A(AttVal) => Instance
      Attributes := Attributes ∪ {A '=' AttVal}
   if ElemOrder = ∅ then
      Tree := '<' Class Attributes '/>'
   else
      for_each Elem in ElemOrder do
         get_slot_desc(slot_desc(Elem,Type,_,_)) => Class
         Value := ∅
         if Type = string then
            if Elem ∈ EmptyAtts then
               for_each get_Elem(ElemVal) => Instance
                  such_that
                     ElemVal = yes
                  do
                     Value := Value ∪ {'<' Elem '/>'}
            elseif Elem \= content then
               for_each get_Elem(ElemVal) => Instance do
                  Value := Value ∪append {'<' Elem '>' ElemVal '</' Elem '>'}
            else
               get_content(ElemVal) => Instance
               Value := {ElemVal}
         elseif (*-Elem) ∉ Alias then
            for_each get_Elem(ElemVal) => Instance do
               Value := Value ∪append {xml_subtree(Elem,ElemVal)}
         else
            for_each get_Elem(ElemVal) => Instance do
               Value := Value ∪append {encaps_subtree(Alias,Elem,ElemVal)}
         Elements := Elements ∪append Value
      Tree := '<' Class Attributes '>' Elements '</' Class '>'
```

```
function encaps_subtree(Aliases,SystemClass,Instance) returns Tree
   get_instance_of(MetaClass) => SystemClass
   if MetaClass = xml_seq then
      Tree:= encaps_seq_subtree(SystemClass,Instance)
   else
      Tree:= encaps_alt_subtree(Aliases,SystemClass,Instance)

function encaps_seq_subtree(SystemClass,Instance) returns Tree
   get_elem_order(ElemOrder) => SystemClass
   get_alias(Alias) => SystemClass
   get_empty(EmptyAtts) => SystemClass
   Elements := ∅
   for_each Elem in ElemOrder do
      get_slot_desc(slot_desc(Elem,Type,_,_)) => SystemClass
      Value := ∅
      if Type = string then
         if Elem ∈ EmptyAtts then
            for_each get_Elem(ElemVal) => Instance
               such_that
                  ElemVal = yes
               do
                  Value := Value ∪ {'<' Elem '/>'}
         elseif Elem \= content then
            for_each get_Elem(ElemVal) => Instance do
               Value := Value ∪append {'<' Elem '>' ElemVal '</' Elem '>'}
         else
            get_content(ElemVal) => Instance
            Value := {ElemVal}
      elseif (*-Elem) ∉ Alias then
         for_each get_Elem(ElemVal) => Instance do
            Value := Value ∪append {xml_subtree(Elem,ElemVal)}
      else
         for_each get_Elem(ElemVal) => Instance do
            Value := Value ∪append {encaps_subtree(Alias,Elem,ElemVal)}
      Elements := Elements ∪append Value
   Tree := Elements

function encaps_alt_subtree(Aliases,SystemClass,Instance) returns Tree
   get_alias(Alias) => SystemClass
   get_empty(EmptyAtts) => SystemClass
   Elements := ∅
   for_each A in Aliases
      such_that
         A = Elem - SystemClass
      do
         get_Elem(ElemVal) => Instance
         if ElemVal ≠ ∅ then
            get_slot_desc(slot_desc(Elem,ElemType,_,_)) => SystemClass
            Value := ∅
            if Type = string then
               if Elem ∈ EmptyAtts then
                  for_each get_Elem(ElemVal) => Instance
                     such_that
                        ElemVal = yes
                     do
                        Value := Value ∪ {'<' Elem '/>'}
               elseif Elem \= content then
                  for_each get_Elem(ElemVal) => Instance do
                     Value := Value ∪append {'<' Elem '>' ElemVal '</' Elem '>'}
               else
                  get_content(ElemVal) => Instance
                  Value := {ElemVal}
            elseif (*-Elem) ∉ Alias then
               for_each get_Elem(ElemVal) => Instance do
                  Value := Value ∪append {xml_subtree(Elem,ElemVal)}
            else
               for_each get_Elem(ElemVal) => Instance do
                  Value := Value ∪append {encaps_subtree(Alias,Elem,ElemVal)}
            Elements := Value
            exit     % for_each loop
   Tree := Elements

function generate_shallow_xml(TopElement,DocOID) returns DOC
   get_children(TopElementOID) => DocOID
   get_elem_order(ElemOrder) => TopElement
   Elem in ElemOrder
   get_att_lst(AttList) => Elem
```

```
   DOC := ∅
   for_each X in get_Elem(X) => TopElementOID do
      Attributes := ∅
      for_each A in AttList do
         get_A(AttVal) => X
         Attributes := Attributes ∪ {A '=' AttVal}
      DOC := DOC ∪append {'<' Elem Attributes '/>'}
   if DOC = ∅ then
      DOC := '<' TopElement '/>'
   else
      DOC := '<' TopElement '>' DOC '</' TopElement '>'

function create_result_schema(TopElement) returns DTD
   Elements := create_element_decl(TopElement)
   DTD := '<!DOCTYPE' ∪ TopElement ∪ '[' ∪ Elements ∪ ']>'

function create_element_decl(Class) returns DTDFragment
   get_elem_order(ElemOrder) => Class
   get_att_lst(AttList) => Class
   get_empty(EmptyAtts) => Class
   get_alias(Alias) => Class
   ATTS := ∅
   SUB_ELEMENT_DECLS := ∅
   if AttList ≠ ∅ then
      for_each A in AttList do
         get_slot_desc(slot_desc(A,AttType,AttCard,AttReq)) => Class
         if AttCard = list then
            Type := NMTOKENS
         else
            Type := CDATA
         if AttReq = mandatory then
            DEF := #REQUIRED
         else
            DEF := #IMPLIED
         ATTS := ATTS ∪ {A Type DEF}
      ATTS := '<!ATTLIST' Class ATTS '>'
   if ElemOrder ≠ ∅ then
      SUB_ELEMENTS := ∅
      for_each Elem in ElemOrder do
         get_slot_desc(slot_desc(Elem,ElemType,ElemCard,ElemReq)) => Class
         if ElemType = string then
            if Elem ∈ EmptyAtts then
               SUB_ELEMENT := Elem cardinality_sign(ElemCard,ElemReq)
               SUB_ELEMENT_DECL := '<!ELEMENT' Elem 'EMPTY>'
            elseif Elem \= content then
               SUB_ELEMENT := Elem cardinality_sign(ElemCard,ElemReq)
               SUB_ELEMENT_DECL := '<!ELEMENT' Elem '(#PCDATA)>'
            else
               SUB_ELEMENT := '#PCDATA'
         elseif (*-Elem) ∉ Alias then
            SUB_ELEMENT := Elem cardinality_sign(ElemCard,ElemReq)
            SUB_ELEMENT_DECL := create_element_decl(Elem)
         else
            SUB_ELEMENT := create_inline_encaps_element_decl(Alias,Elem)
                           cardinality_sign(ElemCard,ElemReq)
            SUB_ELEMENT_DECL := create_encaps_element_decl(Alias,Elem)
         if SUB_ELEMENTS = ∅ then
            SUB_ELEMENTS := SUB_ELEMENT
         else
            SUB_ELEMENTS := SUB_ELEMENTS ∪append {',' SUB_ELEMENT}
         SUB_ELEMENT_DECLS := SUB_ELEMENT_DECLS ∪ SUB_ELEMENT_DECL
      ELEMENT := '<!ELEMENT' Class '(' SUB_ELEMENTS ')>'
   else
      ELEMENT := '<!ELEMENT' Class 'EMPTY>'
   DTDFragment := ELEMENT ATTS SUB_ELEMENT_DECLS

function create_encaps_element_decl(AliasedElems,Class) returns Decl
   get_instance_of(MetaClass) => Class
   if MetaClass = xml_seq then
      get_elem_order(EncapsElements) => Class
   else
      EncapsElements := AliasedElems
   get_empty(EmptyAtts) => Class
   get_alias(Alias) => Class
   SUB_ELEMENT_DECLS := ∅
   for_each Elem in EncapsElements do
      get_slot_desc(slot_desc(Elem,ElemType,ElemCard,ElemReq)) => Class
```

```
      if ElemType = string then
         if Elem ∈ EmptyAtts then
            SUB_ELEMENT_DECL := '<!ELEMENT' Elem 'EMPTY>'
         elseif Elem \= content then
            SUB_ELEMENT_DECL := '<!ELEMENT' Elem '(#PCDATA)>'
      elseif (*-Elem) ∉ Alias then
         SUB_ELEMENT_DECL := create_element_decl(Elem)
      else
         SUB_ELEMENT_DECL := create_encaps_element_decl(Alias,Elem)
      SUB_ELEMENT_DECLS := SUB_ELEMENT_DECLS ∪ SUB_ELEMENT_DECL
   Decl := SUB_ELEMENT_DECLS

function create_inline_encaps_element_decl(AliasedElems,Class) returns Decl
   get_instance_of(MetaClass) => Class
   if MetaClass = xml_seq then
      Sep := ','
      get_elem_order(EncapsElements) => Class
   else
      Sep := '|'
      EncapsElements := AliasedElems
   get_empty(EmptyAtts) => Class
   get_alias(Alias) => Class
   SUB_ELEMENTS := ∅
   for_each Elem in EncapsElements do
      get_slot_desc(slot_desc(Elem,ElemType,ElemCard,ElemReq)) => Class
      if ElemType = string then
         if Elem ∈ EmptyAtts then
            SUB_ELEMENT := Elem cardinality_sign(ElemCard,ElemReq)
         elseif Elem \= content then
            SUB_ELEMENT := Elem cardinality_sign(ElemCard,ElemReq)
         else
            SUB_ELEMENT := '#PCDATA'
      elseif (*-Elem) ∉ Alias then
         SUB_ELEMENT := Elem cardinality_sign(ElemCard,ElemReq)
      else
         SUB_ELEMENT := create_inline_encaps_element_decl(Alias,Elem)
                        cardinality_sign(ElemCard,ElemReq)
      if SUB_ELEMENTS = ∅ then
         SUB_ELEMENTS := SUB_ELEMENT
      else
         SUB_ELEMENTS := SUB_ELEMENTS ∪append {Sep SUB_ELEMENT}
   Decl := '(' SUB_ELEMENTS ')'

function create_shallow_schema(TopElement) returns DTD
   get_elem_order(ElemOrder) => TopElement
   Elem in ElemOrder
   get_slot_desc(slot_desc(Elem,ElemType,ElemCard,ElemReq)) => TopElement
   ELEMENT := '<!ELEMENT' TopElement '(' Elem cardinality_sign(ElemCard,ElemReq) ')>'
   get_att_lst(AttList) => Elem
   ATTS := ∅
   if AttList ≠ ∅ then
      for_each A in AttList do
         get_slot_desc(slot_desc(A,AttType,AttCard,AttReq)) => Elem
         if AttCard = list then
            Type := NMTOKENS
         else
            Type := CDATA
         if AttReq = mandatory then
            DEF := #REQUIRED
         else
            DEF := #IMPLIED
         ATTS := ATTS ∪ {A Type DEF}
      ATTS := '<!ATTLIST' Elem ATTS '>'
   SUBELEMENT := '<!ELEMENT' Elem 'EMPTY >'
   DTD := '<!DOCTYPE' TopElement '[' ELEMENT SUBELEMENT ATTS ']>'

function cardinality_sign(Card,Req) returns SIGN
   if (Card=list) and (Req=mandatory) then
      SIGN := '+'
   elseif (Card=list) and (Req=optional) then
      SIGN := '*'
   elseif (Card=single) and (Req=optional) then
      SIGN := '?'
   else
      SIGN := ''
```