Application examples for SDF links and relations

Application examples for SDF links and relations

This note describes two application use cases for SDF that make use of links and relations.

In both examples, SDF is used in a layered way, to define a core set of models and then apply refinements on those models to define a set of application types, then to define an instance graph or graph template.

The first example is based on a system that models a data flow graph using SDF, then constructs code to implement the graph for a particular embedded platform, for example C++. The nodes in the graph represent physical things like GPIO pins, or internal operations like value normalization or state machine modeling, or communication operations like publishing and subscribing to data streams. When a graph is implemented for an embedded target, each node type is assigned a LWM2M ObjectID and each node in the graph is assigned an InstanceID. Links in the SDF graph are modeled using sdfRelations, and are modeled as LWM2M ObjectLinks for the target graph communication scheme. sdfRelation is used as an abstract model for links, and sdfLink is used as the binding to a concrete model and instance graph.

The second example uses SDF to define an ontology and construct graphs that conform to the ontology-defined constraints. The core SDF model defines an object format with data properties and object properties, or relations. A set of application types are defined with type-specific constraints, and application graphs are constructed from the defined types. The ontology definitions and the application type definitions are used to construct graph skeletons, which are used in lenses to navigate and select sub-graphs, for example when doing validation, knowledge embedding, or for creating domain-specific views of a graph.

DFG node models and graph templates:

This example uses SDF to define a common template for nodes and connections in a DFG, to define some application node types, and to define an application graph template that is constructed from the defined nodes.

The application graph template is then resolved by generating objects for the target runtime and assigning specific routing information to the links. The example implementation targets a C++ runtime system that encapsulates node application logic inside a simple event API.

The implementation example uses the LWM2M information model to configure nodes in the target system. Each application object type is assigned an LWM2M typeID. A set of pre-defined LWM2M Resource Types handle the communication and synchronization. The LWM2M ObjectLink type is used as a DFG connector, with input and output link sub-types defined that use the underlying ObjectLink type. An sdfProperty is used to model each LWM2M ObjectLink in an Object. These are the LWM2M Resource types used in the ObjectFlow communication scheme:

```
// Link types for pull and push data transfer
#define InputLinkType 27000
#define OutputLinkType 27001
// Value types for data connection endpoints
#define InputValueType 27002
#define CurrentValueType 27003
#define OutputValueType 27004
// Timer data types for wrap-around-safe interval activation
#define CurrentTimeType 27005
#define IntervalTimeType 27006
#define LastActivationTimeType 27007
```

Example of layered SDF definition with successive refinement:

sdfRelation can be used to model the linkages between SDF application Objects in the graph:

```
sdfRelation:
  InputLink:
    relType: { const: input }
    target: { sdfType: sdfPointer }
  OutputLink:
    relType: { const: output }
    target: { sdfType: sdfPointer }
```

sdfLink can be used to model the lwm2m links that connect object instances:

```
sdfData:
  ObjectLink:
    type: object
    sdfType: link
    properties:
      TypeID: { sdfRef: /sdfData/IdRange }
      ObjectID: { sdfRef: /sdfData/IdRange }
```

A link property defined in the core model, using sdfRelation as a placeholder for the link data

```
sdfProperty:
  inputLink:
    sdfRef: /sdfData/ObjectLink
    TypeID: /sdfData/TypeID/InputLinkType
    sdfRelation:
      relType: input
```

A link property defined in the graph template as a refinement on the model, adding a target pointer to the linked sdfObject:

```
sdfProperty:
  inputLink:
    sdfRef: /sdfData/ObjectLink
    TypeID: /sdfData/TypeID/InputLinkType
    sdfRelation:
      relType: input
      target: /sdfThing/graph/sdfObject/gpio_10
```

A link property resolved in the graph instance as a refinement of the template, by assigning instance IDs to all of the nodes in the graph, then adding an InstanceID field to the inputLink property, containing the InstanceID of the Object pointed to by the sdfRelation:

```
sdfProperty:
  inputLink:
    sdfRef: /sdfData/ObjectLink
    TypeID: /sdfData/TypeID/InputLinkType
    InstanceID: { const: 13 }
    sdfRelation:
      relType: input
      target: /sdfThing/graph/sdfObject/gpio_10
```

The tools can then map the fully resolved SDF instance definitions to the target configuration, for example C++ header files.

SDF-defined ontology and graph analysis tools

This example uses SDF to define an ontology, which is then used to construct and validate graphs.

How the ontology works

For the ontology, a set of object types (graph nodes) and a set of relation types (graph edges, or predicates) are defined, and the range of relation types each object type may originate (be the subject of), and the corresponding target object types for each relation type, are defined for each object type. In equivalent RDF constructs, the Subject of a triple is the originating object, the Predicate is the relation type, and the Object is the target of the relation.

The object types are defined as sdfObject class with a set of sdfProperty definitions for the Data Properties that object may have. Each sdfObject definition for an object also contains a set of sdfRelation definitions that describe the relations an object may have with other objects by type.

The ontology also defines cardinality constraints, or how many objects may be pointed to by a particular relation. Cardinality uses the sdfRelation minItems and maxItems qualities.

Relations in the ontology may also have sub-relation types that further refine the description of the relationship. For this, the example defines the "relationSubType" property.

The example also adds the "targetType" property to contain a set of object types this relationType may point to. Use of an array to define the set here is shorthand for separate sdfRelation definitions, each having exactly one target type chosen from the set.

```
sdfRelation:
  graphRelation:
    relType: { sdfRef: /sdfData/relationTypes }
    target: { sdfType: sdfPointer }
    property:
      type: object
      properties:
        relationSubType: { sdfRef: /sdfData/relationSubTypes }
        targetType:
          type: array
          items: { sdfRef: /sdfData/objectTypes }

sdfData:
  relationTypes:
    sdfChoice:
      contains: {}
      partOf: {}
      connectedTo: {}
      connectedFrom: {}

  relationSubTypes:
    sdfChoice:
      primary: {}
      secondary: {}
      parent: {}
      child: {}
      next: {}
      previous: {}

  objectTypes:
    sdfChoice:
      System: {}
      Equipment: {}
      Connection: {}
      ControlQuantity: {}
      Media: {}
      Property: {}
```

An example ontological definition for an Object type:

```
sdfObject:
  Equipment:
    sdfRef: /sdfObject/graphObject
    sdfProperty:
      equipmentType:
        sdfRef: /sdfProperty/dataProperty
        type: string
    sdfRelation:
      ControlQuantity:
        sdfRef: /sdfRelation/graphRelation
        relType: contains
        minItems: { const: 1 }
        maxItems: { const: 2 }
        property:
          relationSubType:
            - primary
            - secondary
          targetType:
            const:
              - /sdfObject/ControlQuantity
      InputConnection:
        sdfRef: /sdfRelation/graphRelation
        relType: connectedFrom
        minItems: { const: 1 }
        maxItems: { const: 2 }
        property:
          relationSubType:
          targetType:
            const:
              - /sdfObject/Equipment
      OutputConnection:
        sdfRef: /sdfRelation/graphRelation
        relType: connectedTo
        minItems: { const: 1 }
        property:
          relationSubType:
          targetType:
            const:
              - /sdfObject/Equipment
```

The example system uses a shorthand text format for relations: <relType>.<relationSubType>.<target type>

For example, one of the relations in the example above might me written "contains.primary.ControlQuantity" and would point to the primary ControlQuantity object for this Equipment object. RDF predicate strings can be created using similar notation.

How the ontology is used

A JSON-Schema document is programmatically created from the core definitions that enables a JSON form editor to be used to build new object types by simply filling out a form.

A JSON-Schema document is built containing all of the defined object types that enables a JSON form editor to be used to configure and build graphs from the defined object types by simply filling out items in a form.

To create a graph, a set of objects are configured with data properties, and the sdfRelation "target" qualitiy values are configured with sdfPointers to the desired objects in the graph. The resulting structure can then be further processed to load into a graph database or other processing.

In the example system, graph validation and navigation tool uses a bound lens pattern to navigate and factorize the graph based on JSON control documents that constrain paths through the graph. A state-machine filter examines each object using a type-specific state/next state construct, and filters its data properties and its relations to decide which relations to follow for the next object/state. The resulting output graph is a factorization of the input graph based on the paths selected by the lens document. The lens document can be said to be a skeleton of the output graph.

In the example system, a visualization tool is used to render plantUML object diagrams of the selected graphs for documentation and manual validation through inspection.

In the example system, object properties are used to embed knowledge into the graph for application processing.

Last updated 2023-01-28 17:17:07 -0800