Template-based constraint management

To automate constraint generation and also the required updating process, we use templates in which the constant parts of a constraint family as well as the variability are defined.

Template:

- Instantiation context (IC)
- Abstract constraint expression (ACE)
- Variable definition (VD)
- Instantiation information (II)
- Data extraction expressions (DEE)

The IC defines an element or a pair of elements for which the template should be instantiated.
In the ACE the desired constraint is defined. Variables are used to replace variable parts of the constraints.
These variables must be defined explicitly in the VD.
The II defines how instances of the elements defined in the IC can be used in the DEEs (i.e., reference names for passed elements to be used in the DEE).
The DEEs use the elements and their reference names from the II to perform the extraction of data that is then inserted into the ACE to generate the desired constraint.

In the following section we present several templates we have previously used.
Please note that these templates are defined using the template-based constraint management approach and that the XLM requires a slightly different form. In particular, the templates presented here use OCL expressions while the XLM currently requires templates to be written in the internal language (i.e., Abstract Rule Language (ATL)) of the used consistency checker Model/Analyzer.

Domain: Component-oriented modeling

In the domain of component-oriented modeling, we have a metamodel that defines a Component class and a Communication class. Components can have an arbitrary number of communications assigned through a reference called com. Each Communication must provide exactly one target and one source component. The derived reference inv is used to retrieve all components involved in a commonication (i.e., the components reached through target and source).
Components can have an arbitrary number of sub-components, reach through the reference "sub".
In addition to communications, components can also use other components directly without the overhead of a full-blown communication. These used components are provided through a reference called "use".

Direct communication (i.e., connecting components through the "use" reference) is only allowed between components that are both sub-components of the same top-level component (i.e., they are within the same "domain").

Templates:

References

IC: <Class, Reference>

ACE: <context C inv: self.R->size()>=MIN and self.R->size()<=MAX>

VD: <C, R, MIN, MAX>

II: <Class c, Reference r>

DEE: <c.name, r.name, r.min, r,max>

This template generated constraints that check the correct number of elements connected through references. For example that communications actually provide only 1 target and source. Changing the maximum number of targets, for example to 100, leads to an update of the generated constraint.

Derived Reference

IC: <Class, DerivedReference>

ACE: <context C inv self.DR->includesAll(REFS->collect(x|self.{x}))>

VC: <C, DR, REFS>

II: <Class c, DerivedReference dr>

DEE: <c.name, dr.name, dr.refs->collect(name)>

This template uses the special notation self.{x}. When the template is instantiated, REFS is replaces with a set of String objects that represent the names of the references aggregated through the derived reference. The collect-expression executed on this set means that the variable x is replaced with Strings. With self.{x}, we can use such a String and retrieve the elements reached through a connection with a name equal to the String. In summary, the elements returned by the derived reference must include all elements reached through the aggregated references.

Domain

IC: <Component>

ACE: <DOM-includesAll(self.use)>

VD: <DOM>

II: <Component c>

DEE: <domain(self)>

where "domain(Component) is a function that calculates the entire domain of a component. When the template is instantiated, the domain for the component is calucated and the constraint requires that all elements used by the component are within the domain. This means that a domain does not have to be re-calculated when the set of used components is changed. Only changes in the domain lead to an actual re-calculation of the domain and therefore calculation effort is reduced.

Domain: Feature models (see Paper)

In the domain of feature models we defined templates for a simple, yet complete feature model metamodel. These templates generate constraints from a feature model that then check specific product configurations.

Metamodel:

The metamodel consists of Feature elements that can be linked by Association elements. An association has an arbitrary number of targets, which have to be other features. There are several different associationas available: Required, Or, Xor, and Optional.

Required associations are used to express that the source feature requires all target features to be present in a configuration.

Or associations express that at least one of the target features must be selected.

Xor associations require exactly one of the targets to be present in a configuration.

Optional associations are used to express that the target features may or may not be part of a configuration.

Templates:

Required

IC: <Feature, Required>

ACE: <context FEATURE inv: self.allReferences->includesAll(TARGETS)>

VD: <FEATURE, TARGETS>

II: <Feature f, required r>

DEE: <f.name, r.targets->collect(name)>

The feature must provide a reference with the name of the required sub-feature to a non-null element for all required sub-features.

Or

IC: <Feature, Or>

ACE: <context FEATURE inv: not self.allReferences->excludesAll(TARGETS)>

VD: <FEATURE, TARGETS>

II: <Feature f, Or o>

DEE: <f.name, o.targets->collect(name)>

The list of references from the feature to sub-features contains at least one reference that matches a possible target.

Xor

IC: <Feature, Xor>

ACE: <context FEATURE inv: self.allReferences->select(n|TARGETS->includes(n))->size()=1>

VD: <FEATURE, TARGETS>

II: <Feature f, Xor x>

DEE: <f.name, x.targets->collect(name)>

The number of references that match possible targets of the Xor-association must be exactly 1.

Please note that there are several ways to write OCL expressions with equal semantics. For example, we could have used to ACE of the Xor-template and change the required size to be greater than zero instead of being exactly one for the Or-template.

Domain: General purpose modeling with ECore

For general purpose modeling we used the ECore metamodel and defined templates that checked structural integrity of models.

Metamodel:

The ECore metamodel consists of three main elements: EClass, EAttribute, and EReference.

An EClass is used to model entities (e.g., the UML elements Class, Message, or Transition).

An EClass object can have an arbitrary number of structural features. EAttribute and EReference objects are such structural features that can be added to an EClass. These elements have cardinality values (i.e., lowerBound and upperBound) and define the type of the reached elements (eType) which have to be EClass objects.

Templates:

EAttribute

IC: <EClass, EAttribute>

ACE: <context NAME inv: self.ATTRIBUTE->size()>=MIN and self.ATTRIBUTE->size<=MAX and self.ATTRIBUTE->forAll(e|e instanceof TYPE)>

VD: <NAME, ATTRIBUTE, MIN, MAX, TYPE>

II: <EClass c, EAttribute a>

DEE: <c.name, a.name, a.lowerBound, a.upperBound, a.eType>

For an instance of EClass (e.g., the UML element Class) and an associated instance of EAttribute (e.g., the UML Attribute isActive for the UML element Class) the template generates a constraint that checks that:

i) Instances of the modeled element provide the right number of elements (between MIN and MAX)

ii) the reached elements have the correct type.

For example, this template generates the following constraint for the EClass "Class" and the EAttribute "isActive":

context Class inv: self.isActive->size()>=1 and self.isActive->size()<=1 and self.isActive()->forAll(e|e instanceof Boolean)

This constraint is then enforced on all instances of Class that are generated during UML modeling and ensures that all modeling elements representing UML Class instances provide the required information.

EReference

IC: <EClass, EReference>

ACE: <context NAME inv: self.REFERENCE->size()>=MIN and self.REFERENCE->size<=MAX and self.REFERENCE->forAll(e|e instanceof TYPE)>

VD: <NAME, ATTRIBUTE, MIN, MAX, TYPE>

II: <EClass c, EReference r>

DEE: <c.name, r.name, r.lowerBound, r.upperBound, r.eType>

Domain: UML modeling

Metamodel: We use the UML metamodel to define templates that check instance of the UML metamodel for proper use.

Instance

IC: <Instance>

ACE: <context Package inv: self.classes->exists(c|c.name=CLASSNAME)>

VD: <CLASSNAME>

II: <Instance i>

DEE: <i.className>

For Instance objects the generated constraints ensure that there is a Class in the Package that has the name used by the instance.

Message

IC: <Message>

ACE: <context Class inv self.name=RNAME implies self.providedMethods->exists(m|m.name=MNAME)>

VD: <RNAME, MNAME>

II: <Message m>

DEE: <m.receiver.className, m.name>

A class that was used as the receiver in a message must provide a method with a name matching the name of the message.

Transition

IC: <Transition>

ACE: <context class inv: self.name=SNAME implies self.providedMethods->exists(m|m.name=MNAME)>

VD: <SNAME, MNAME>

II: <Transition t>

DEE: <t.source.class.name, t.name>

A transition name must have a matching method provided by the class belonging to the source state of the transition.