Changing an ArchiMate element type in your model

It may sometimes be necessary to change an ArchiMate element type into another type. This can be either due to incorrect use of an element type to begin with, or can be caused by historical errors. Instead of having to change each instance of the element type, you can use a script that allows you to replace an element with another type without having to delete or to recreate all the associated relationships and locations where the element has been used. Before you start using the script, please take note of the following:

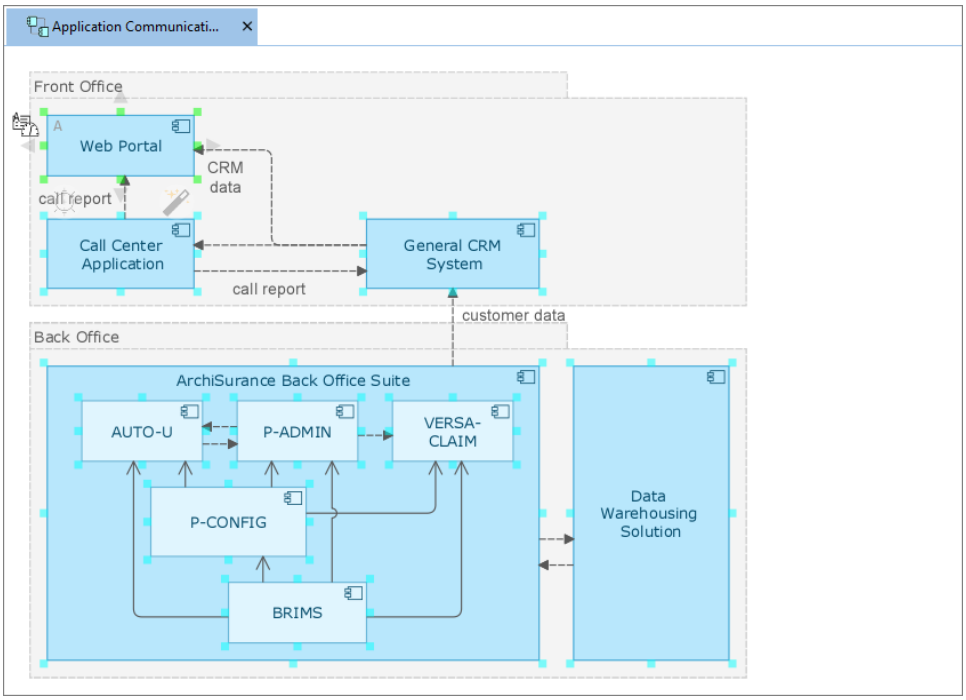
- The script does not take any internal object ID's into account. These values are not preserved.
- The script does not work with e.g. portfolios in which the element to be changed occurs, since a new object is created.
- Changing an element from one type of one layer (for example a Business Service) to the same type of another layer (for example an Application Service) is possible. However, the associated change of color of the element will not be shown on the element in the views. The color of the new type will not be visible until you add the element new onto a view.

The script is shown further down the page.

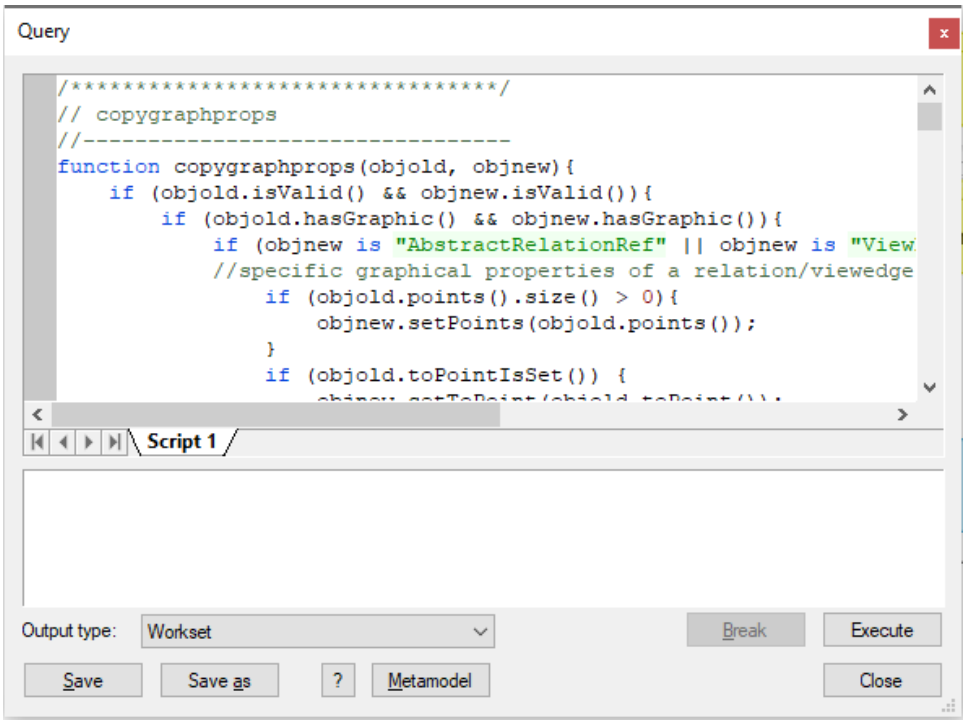
> It is recommended to test run the script on an example model package or make a backup of the production model package before starting to use the script for real.

Steps:

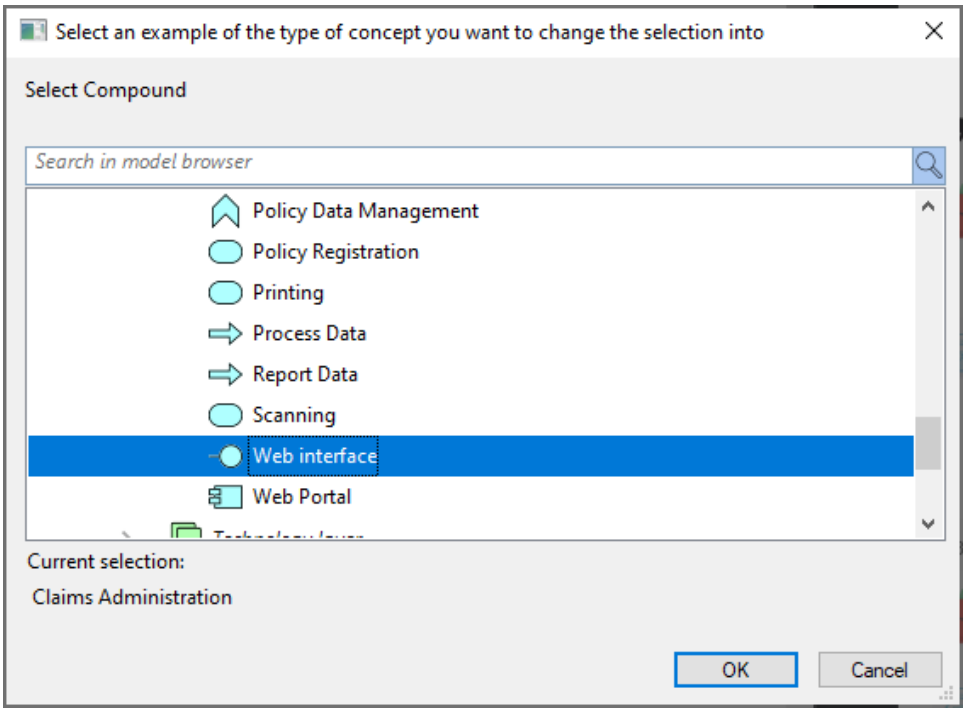
1. Select the element(s) that you want to change. You can select them in a view or in the model browser. If you select multiple elements of the same type, they will all be changed into the new element type. In the example below we want to change the application components into application interfaces.



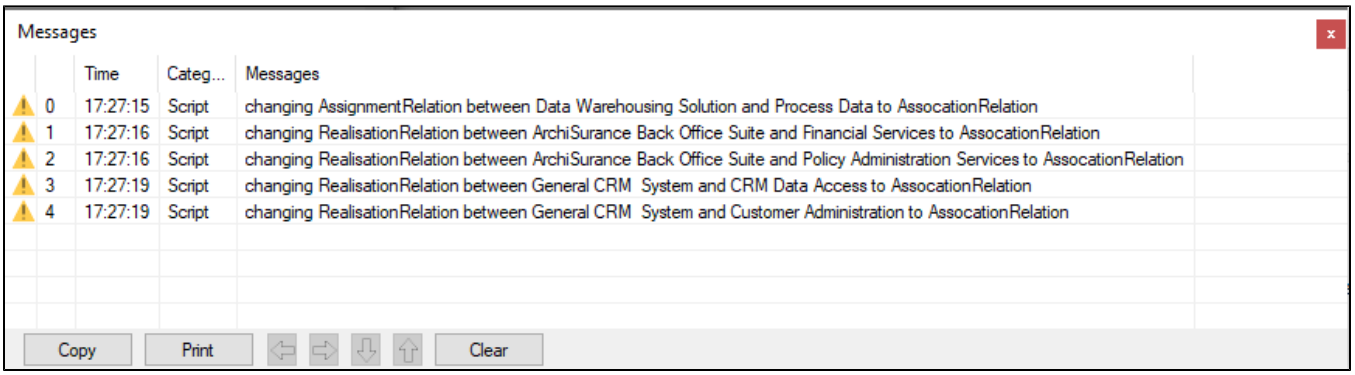
2. Open the Query editor (Ctrl+Q) and past the script into the window. Set the output type to 'Workset' and execute the script.



3. In the window that appears, select an element that represents the desired new element type. The script will change the earlier selected element(s) to new elements of that type. For the example it would be an application interface (any).

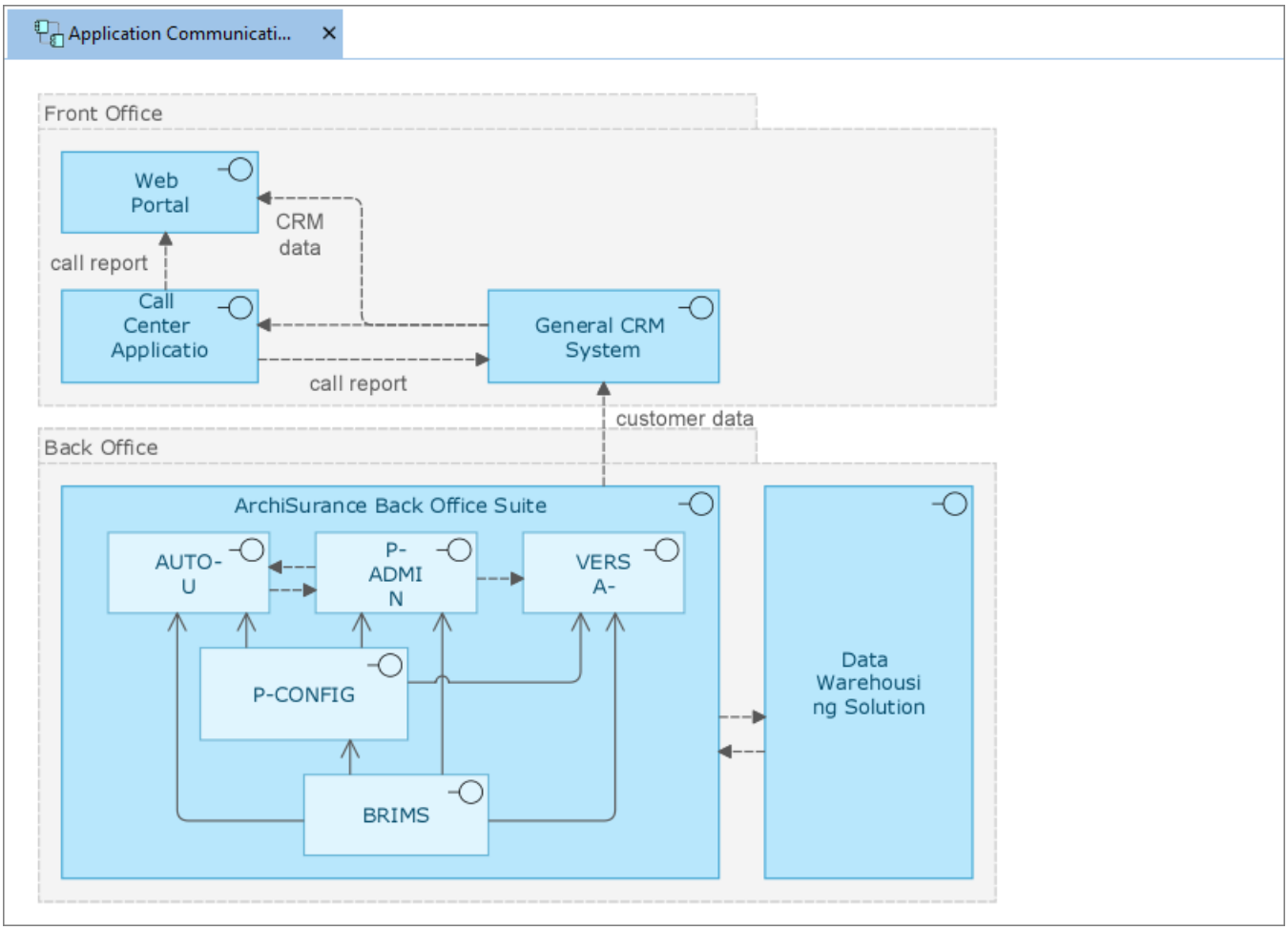


The script will try to preserve profiles, (multilingual) attributes, metric values and relationships as much as possible. Illegal relationships are converted to directed associations. The Messages windows shows the progress of the change, informs you about what happens and notes any findings during the process.



The script will output a workset with all the new elements.

In your model, the changes will also be visible. In the model browser and in the view. Following the example, the application components are now application interfaces:



Script

```
/*********************************/
// copygraphprops
//---------------------------------
function copygraphprops(objold, objnew){
        if (objold.isValid() && objnew.isValid()){
                if (objold.hasGraphic() && objnew.hasGraphic()){
                        if (objnew is "AbstractRelationRef" || objnew is "ViewEdge"){
                        //specific graphical properties of a relation/viewedge
                               if (objold.points().size() > 0){
                                       objnew.setPoints(objold.points());
                               }
                               if (objold.toPointIsSet()) {
                                       objnew.setToPoint(objold.toPoint());
                               }
                               if (objold.fromPointIsSet()) {
                                       objnew.setFromPoint(objold.fromPoint());
                               }
                               fs = objold.fromLineSegment();
                               if(fs > 0) {
                                       objnew.setFromLineSegment(fs);
                                       fp = objold.fromLineSegmentPercentage();
                                       objnew.setFromLineSegmentPercentage(fp);
                               }
                               ts = objold.toLineSegment();
                               if(ts > 0) {
                                       objnew.setToLineSegment(ts);
                                       tp = objold.toLineSegmentPercentage();
                                       objnew.setToLineSegmentPercentage(tp);
```

```
                }
        }
        else {
        //specific graphical properties of an object:
                objnew.setOrigin(objold.origin());
                if (!objold.dimensions().empty()) {
                        objnew.setDimensions(objold.dimensions());
                }

                if (objold.isCollapsed()) {
                        objnew.collapse();
                }
        }

        //parse all possible graphical properties
        if (objold.lineTransparencyIsSet() ) {
                objnew.setLineTransparency(objold.lineTransparency());
        }
        if (objold.fillTransparencyIsSet() ) {
                objnew.setFillTransparency(objold.fillTransparency());
        }
        if (objold.textTransparencyIsSet() ) {
                objnew.setTextTransparency(objold.textTransparency());
        }
        if (objold.fontIsSet() ) {
                objnew.setFont(objold.font());
        }
        if (objold.fontBoldIsSet() ) {
                objnew.setFontBold(objold.fontBold());
        }
        if (objold.fontItalicIsSet() ) {
                objnew.setFontItalic(objold.fontItalic());
        }
        if (objold.fontUnderlineIsSet() ) {
                objnew.setFontUnderline(objold.fontUnderline());
        }
        if (objold.fontSizeIsSet() ) {
                objnew.setFontSize(objold.fontSize());
        }
        if (objold.foregroundColorIsSet() ) {
                objnew.setForegroundColor(objold.foregroundColor());
        }
        if (objold.backgroundColorIsSet() ) {
                objnew.setBackgroundColor(objold.backgroundColor());
        }
        if (objold.fontColorIsSet() ) {
                objnew.setFontColor(objold.fontColor());
        }
        if (objold.iconVisibilityIsSet() ) {
                objnew.setIconVisibility(objold.iconIsVisible());
        }
        if (objold.labelDirectionIsSet() ) {
                objnew.setLabelDirection(objold.labelDirection());
        }
        if (objold.labelOffsetIsSet() ) {
                objnew.setLabelOffset(objold.labelOffset());
        }
        if (objold.labelPositionIsSet() ) {
                objnew.setLabelPosition(objold.labelPosition());
        }
        if (objold.lineStyleIsSet() ) {
                objnew.setLineStyle(objold.lineStyle());
        }
        if (objold.lineWidthIsSet() ) {
                objnew.setLineWidth(objold.lineWidth());
        }
        if (objold.shapeIsSet() ) {
                objnew.setShape(objold.shape());
        }
}
Component.utils:invalidate(objnew);
```

```
        }
        return undefined;
}

/*******************************/
// findBaseRelType
//-------------------------------
function findBaseRelType (RelType) {
        nameSplit = "";
        if (RelType.isString()) {
                nameSplit = TypeID(RelType).name().split(":");
        }
        else if (RelType.isTypeID()) {
                nameSplit = RelType.name().split(":");
        }
        mm = nameSplit[1];
        typName = nameSplit[2];
        // RelType = RelType.replace("ArchiMate:", "");
        c = InternalObject("configuration").context(modelpackage);
        inType = c.metaModel(mm).concept(typName);
        if (inType.isAbstract()) {
                return undefined;
        }
        else {
                basType = inType.baseConcept();
                if (basType.isAbstract()) {
                        return inType.name();
                }
                lasType = basType;
                while (!basType.isAbstract()) {

                        lasType = basType;
                        basType = basType.baseConcept();

                }
                return lasType.name();
        }
}

/***************************************/
// deconstruct
// removes references to oldObj
// and adds references to newObj instead
//---------------------------------------
function deconstruct(obj, atnm, refAttr, oldObj, newObj) {
        if (refAttr.isObject()) {
                if(refAttr == oldObj) {
                        return newObj;
                } else {
                        return refAttr;
                }
        } else if (refAttr.isLink()) {
                if(refAttr.object == oldObj) {
                        newLink = Link(newObj);
                } else {
                        newLink = refAttr;
                }
                return newLink;
        } else if (refAttr.isStructure()) {
                newStruct = Structure();
                forall field, value in r {
                        if(value == oldObj) {
                                newStruct.add(field, newObj);
                        } else {
                                newStruct.add(field, value);
                        }
                }
                return newStruct;
        } else if (refAttr.isPair()) {
                if(refAttr.first == oldObj) {
                        newPair = Pair(newObj, refAttr.second);
```

```
                } else if(refAttr.second == oldObj) {
                        newPair = Pair(refAttr.first, newObj);
                } else {
                        newPair = refAttr;
                }
                return newPair;
        } else if (refAttr.isList() || refAttr.isSet()) {
                newCollection = refAttr.isList() ? List() : Set();
                forall r in refAttr {
                        newVal = deconstruct(obj, atnm, r, oldObj, newObj);
                        if(!newCollection.contains(newVal)) // this assumes we don't want duplicate entries
in lists
                                newCollection.add(newVal);
                }
                return newCollection;
        } else { // some other type
                warning format("type of %s not supported yet"), refAttr.toString();
                return refAttr;
        }
}

/*******************************************/
// copyAttr
// copies attribute and metric values
// trying to be smart about the types of values
//-----------------------------------------
function copyAttr(oldObject, newObject, attr) {
        prof = attr.profile();
        if(prof != BasicProfile()) {
                pn = prof.name();
                if(!newObject.hasProfile(pn)) {
                        if(newObject.canAssignProfile(pn)) {
                                newObject.assignProfile(pn);
                        } else {
                                warning format("cannot assign profile %s to %s", pn, newObject);
                                return false;
                        }
                }
        }

        if(!newObject.hasAttr(attr)) {
                warning format("%s does not have attribute %s", newObject.name(), attr.name());
                return false;
        }

        forall "MotivationMetric" metric in modelpackage {
                value = undefined;
                ok = ArchiMate.metrics:metricValue(oldObject, metric, value);
                if(!value.isUndefined()) {
                        ArchiMate.metrics:setMetricValue(newObject, metric, value);
                }
        }

        changed = false;
        forall "MM_Language" ml in modelpackage {
                lang = ml.toString();
                oldVal = oldObject.hasUserValueRef(attr, lang) ? oldObject.attrValue(attr, lang) : undefined;
                if(oldVal != undefined) {
                        newObject.setAttrValue(attr, oldVal, lang);
                        changed = true;
                }
        }
        return changed;
}

/*************************************************************/
// addNewRelation
// adds new relation, copying attributes from an old one
// if there already is such a relationship, return that instead
//-------------------------------------------------------------
function addNewRelation(srcObj, destObj, rel) {
```

```
        newRelType = findBaseRelType(rel.type());
        exists = undefined;
        // check whether we already have a relationship like this
        forall r in srcObj.relationsFrom() {
                if( (r.to == destObj) &&
                    (findBaseRelType(r.type()) == newRelType) &&
                    (r.name() == rel.name()) ) {
                        exists = r;
                }
        }

        if(exists != undefined) {
                newRel = exists;
        } else {
                if(srcObj.canAddRelationTo(destObj, newRelType)) {
                        newRel = srcObj.addRelationTo(destObj, newRelType);
                } else {
                        warning "changing " + newRelType + " between " + srcObj.name() +
                                        " and " + destObj.name() + " to AssocationRelation";
                        newRelType = "AssociationRelation";
                        newRel = srcObj.addRelationTo(destObj, newRelType);
                        newRel.setAttrValue("isDirected", true);
                }
        }

        forall attr in rel.attrs() {
                copyAttr(rel, newRel, attr);
        }

        relRels = rel.relations();
        forall relRel in relRels {
                if(relRel.from == rel) {
                        newRelRel = newRel.addRelationTo( relRel.to, findBaseRelType(relRel.type()) );
                } else { // relRel.to == rel
                        newRelRel = relRel.from.addRelationTo(newRel, findBaseRelType(relRel.type()) );
                }
                forall attr in newRelRel.attrs() {
                        copyAttr(relRel, newRelRel, attr);
                }
        }

        return newRel;
}

/*****************************************/
// addNewRelationRef
// adds new graphical reference for a relation,
// copying graphical properties from an old one
//----------------------------------------
function addNewRelationRef(relRef, newRel, from, to) {
        newRelRef = undefined;
        relRefPar = relRef.parent();
        if ( relRefPar.canAddNewReference(newRel, from, to) ) {
                newRelRef = relRefPar.addNewReference(newRel, from, to);
                copygraphprops(relRef, newRelRef);
        } else {
                relRefView = relRef.parent("AbstractView");
                if ( relRefView.canAddNewReference(newRel, from, to) ) {
                        newRelRef = relRefView.addNewReference(newRel, from, to);
                        newRelRef.moveTo(relRefPar);
                        copygraphprops(relRef, newRelRef);
                } else {
                        warning format("Could not add reference to relation %s to view '%s' of type %s",
                                                newRel.toString(), relRefView, relRefView.type());
                        return;
                }
        }

        oldRelRels = relRef.to.relations();
        newRelRels = newRel.relations();
        forall relRel in oldRelRels {
```

```
                matchRel = undefined;
                forall nr in newRelRels {
                        if( (nr.from == relRel.from && nr.to == newRel) ||
                            (nr.from == newRel && nr.to == relRel.to) ) {
                                matchRel = nr;
                        }
                }
                forall relRelRef in relRel.references() {
                        if(relRelRef.fromReference() == relRef) {
                                addNewRelationRef(relRelRef, matchRel, newRelRef, relRelRef.toReference());
                        } else if(relRelRef.toReference() == relRef) {
                                addNewRelationRef(relRelRef, matchRel, relRelRef.fromReference(), newRelRef);
                        } else error "relation reference not found: " + relRelRef.toString();
                }
        }

}

/*******************************/
// cloneObject
//--------------------------------
function cloneObject(oldObject, newObject) {
        // merge profile attributes
        forall attr in oldObject.attrs() {
                copyAttr(oldObject, newObject, attr);
        }

        // copy all profile references to newObject and remove references to oldObject
        forall at, objs in oldObject.profileReferrals() {
                forall obj in objs {
                        atnm = at.toString();
                        refAttr = obj.attrValue(atnm);
                        newVal = deconstruct(obj, atnm, refAttr, oldObject, newObject);
                        obj.setAttrValue(atnm, newVal);
                }
        }
}

/*******************************/
// cloneRelations
//--------------------------------
function cloneRelations(byref objIndex, byref relIndex) {
        forall oldObject, newObject in objIndex {
                forall rel in oldObject.relations() {
                        if(relIndex.valueFor(rel) == undefined) {
                                if(rel.from == oldObject) {
                                        newEnd   = objIndex.valueFor(rel.to);
                                        if(newEnd == undefined) {
                                                newEnd = rel.to;
                                        }
                                        newRel = addNewRelation(newObject, newEnd, rel);
                                        relIndex.add(rel, newRel);
                                } else { // rel.to must be oldObject
                                        newStart = objIndex.valueFor(rel.from);
                                        if(newStart == undefined) {
                                                newStart = rel.from;
                                        }
                                        newRel = addNewRelation(newStart, newObject, rel);
                                        relIndex.add(rel, newRel);
                                }
                        }
                }
        }
}

/*******************************/
// cloneGraphics
// copy all graphical references
//--------------------------------
function cloneGraphics(byref objIndex, byref relIndex) {
        refIndex = Index();
```

```
        // create new references and copy all properties from oldones
        forall oldObject, newObject in objIndex {
                forall ref in oldObject.references() {
                        refPar = ref.parent();
                        if(refPar.canAddNewReference(newObject)) {
                                newRef = refPar.addNewReference(newObject);
                                copygraphprops(ref, newRef);
                                refIndex.add(ref, newRef);
                        } else {
                                warning format("Could not create reference to element '%s' of type %s on
view '%s' of type %s",
                                                                newObject.name(), newObject.type(), refPar.name(),
refPar.type());
                        }
                }
        }

        //move all refchildren to the new references
        forall ref, newRef in refIndex {
                forall childRef in ref.children() {
                        if(childRef is "AbstractRelationRef") {
                                oldPoints = childRef.points();
                        } else {
                                oldOrig = childRef.origin();
                        }
                        childRef.moveTo(newRef);
                        if(childRef is "AbstractRelationRef") {
                                childRef.setPoints(oldPoints);
                        } else {
                                childRef.setOrigin(oldOrig);
                        }
                }
        }
        //add all existing relations from old object to new object
        forall oldRel, newRel in relIndex {
                forall relRef in oldRel.references() {
                        startRef = refIndex.valueFor(relRef.fromReference());
                        endRef   = refIndex.valueFor(relRef.toReference());
                        if(startRef.isUndefined()) {
                                startRef = relRef.fromReference();
                        }
                        if(endRef.isUndefined()) {
                                endRef = relRef.toReference();
                        }
                        addNewRelationRef(relRef, newRel, startRef,
endRef);
                }
        }
}

/*********************************************/
// createObject
// creates a clone object of obj with another type
//-------------------------------------------
function createObject(obj, type) {
        par = obj.parent();
        if(par.canAddNewObject(type)) {
                newObj = par.addNewObject(type);
        } else {
                mod = obj.parent("MM_Model");
                par = mod.getSchemeFor(type, true);
                newObj = par.addNewObject(type);
        }
        return newObj;
}

/*********************************************/
// moveChildren
// moves future orphans to their new parent
//-------------------------------------------
function moveChildren(oldObject, newObject) {
```

```
	forall child in oldObject.children() {
		if(child.canMoveTo(newObject)) {
			child.moveTo(newObject);
		} else {
			mod = child.parent("MM_Model");
			par = mod.getSchemeFor(child.type(), true);
			child.moveTo(par);
		}
	}

}

/************************************************/
// fixNesting
// preserve the model browser nesting structure
//-----------------------------------------------
function fixNesting(objIndex) {
	forall removeObj, keepObj in objIndex {
		oldPar = removeObj.parent();
		newPar = objIndex.valueFor(oldPar);
		if(!newPar.isUndefined() && newPar.isValid()) {
			if(keepObj.canMoveTo(newPar)) {
				keepObj.moveTo(newPar);
			}
		}
	}

	forall removeObj, keepObj in objIndex {
		if(removeObj.isValid()) {
			moveChildren(removeObj, keepObj);
			removeObj.delete();
		}
		output keepObj;
	}
}

//-------------------------
// START SCRIPT
/*************************/

if (selection.isList() && !selection.empty()) {
	exampleObj = ask("AbstractCompound", "Select an example of the type of concept you want to change
the selection into");
	if(exampleObj == undefined) {
		stop;
	}
	objIndex = Index();
	forall obj in selection {
		if(obj is "Reference") {
			obj = obj.to;
		}
		if(!obj.isObject() || !obj.isValid()) {
			error "selection invalid";
			stop;
		}
		if(obj.type() != exampleObj.type()) {
			newObj = createObject(obj, exampleObj.type());
			objIndex.add(obj, newObj);
			cloneObject(obj, newObj);
		}
	}

	relIndex = Index();
	cloneRelations(objIndex, relIndex);
	cloneGraphics(objIndex, relIndex);
	fixNesting(objIndex);
} else {
	error "nothing selected / execution canceled";
}
```

Related articles

- Adding content to an ArchiMate view
- Model content checks for ArchiMate models