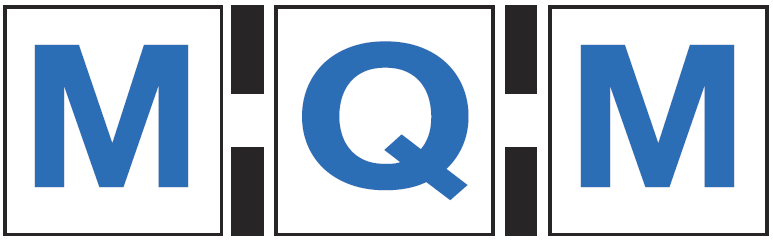
G+DE – Forward Completion Design

*P. -Struss*

21.04.2020

Version 2



**Contents**

[Document History 3](#_gjdgxs)

[0 Purpose 4](#_1fob9te)

[1 Foundations 5](#_3znysh7)

[1.1 General Assumptions 5](#_2et92p0)

[1.2 Representation of Situation 5](#_tyjcwt)

[2 Forward Completion - Toplevel 6](#_3dy6vkm)

[2.1 Basic Idea of the Algorithm 6](#_1t3h5sf)

[2.2 Pseudo Code Library.ForwardCompletionStructural 7](#_2s8eyo1)

[3 Computing RoleBindings 10](#_17dp8vu)

[3.1 Underlying Ideas 10](#_3rdcrjn)

[3.2 Tables 10](#_26in1rg)

[3.2.1 LibraryTypeTable 10](#_lnxbz9)

[3.2.2 RoleRelationTable 11](#_35nkun2)

[3.2.3 TypeSubsumptionTable 12](#_1ksv4uv)

[3.2.4 SituationTypeTable 13](#_44sinio)

[3.2.5 SituationRelationTable 13](#_2jxsxqh)

[3.3 Operations 13](#_z337ya)

[3.3.1 Determining Relevant Processes and RoleBindings via Types 13](#_3j2qqm3)

[3.3.2 RoleBindings via Relations 13](#_1y810tw)

[3.3.3 Computing Complete and Consistent RoleBinding 13](#_4i7ojhp)

[3.4 StructuralElementType.ProcessTypePreconditions 15](#_2xcytpi)

[3.5 ProcessType.NewInstancesStructural 15](#_1ci93xb)

[3.6 Situation.EffectStructuralElements 15](#_3whwml4)

# Document History

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Changes | Author |
| 1.0 | 6.4.20 | 1st draft, top-level algorithm only | Struss |
| 2 | 21.4.20 | Extensions: quick access to relevant process types, computing role bindings | Struss |

# Purpose

To propose the software design of a first forward completion algorithm (which means a focus on structural aspects only). This version contains the top-level algorithm and ideas about the computation of role bindings for the structural conditions. Missing: instantiation of the structural effects, Properties of relations (e.g. uniqueness) and domain-specific interrelations between them.

# Foundations

## General Assumptions

A number of assumptions are made throughout the document (and it will be pointed out where they are used). They basically state that the Processtype Library and the Type Hierarchy are well-formed and that the representation of a situation is complete and compliant with the Ontology and in itself. This means, in particular,

1. All ObjectRoles and RoleRelations have a type specified in the TypeHierarchy.
2. RoleRelations are properly defined, i.e. the ObjectTypes in the signature of their RelationType are super types of the types associated with the ObjectRoles the roleRelation is defined on
3. The definitions of RoleRelations in a ProcessType are consistent with properties of RelationTypes and their interrelations (e.g. no mutually exclusive relations are defined)
4. The TypeHierarchies for Objects and Relations is an acyclic directed graph.
5. If RelationType RT1 is a specialization of another one, RT2, then the ObjectTypes in its signature are not supertypes of the respective ObjectTypes in the signature of RT2.
6. Situations are compliant with the ontology (e.g. all objects and relations have types from the TypeHierarchy)
7. Situations are consistent (e.g. do not introduce contradictory relations among objects)

## Representation of Situation

The following concepts are restricted to the structural aspects; for instance, existing range restrictions on attributes of StructuralElements are ignored.

A **Situation** (see Fig. 1) is characterized by

* A collection of **ProcessInstances** and
* A collection of **StructuralElements** (instances).

A **ProcessInstance** has

* A **ProcessType** and
* A set of **RoleBindings**, which is complete w.r.t. the StructuralElementsRoles (i.e. ObjectRoles and ObjectRoleRelations) that occur in StructuralConditions or StructuralEffects of the Processtype

A RoleBinding associates a StructuralElement with a StructuralElementsRole, such that it is consistent with the properties of the types. I.e., the StructuralElementType of the former is a subtype of the one of the latter, the uniqueness property has to be respected (an Object cannot be located in different SpatialObjects, etc.)

It may be the case that the situation contains StructuralElements that do not occur in RoleBindings of ProcessInstances.

Note that the extension of a situation is itself a situation according to the definition above.

# Forward Completion - Toplevel

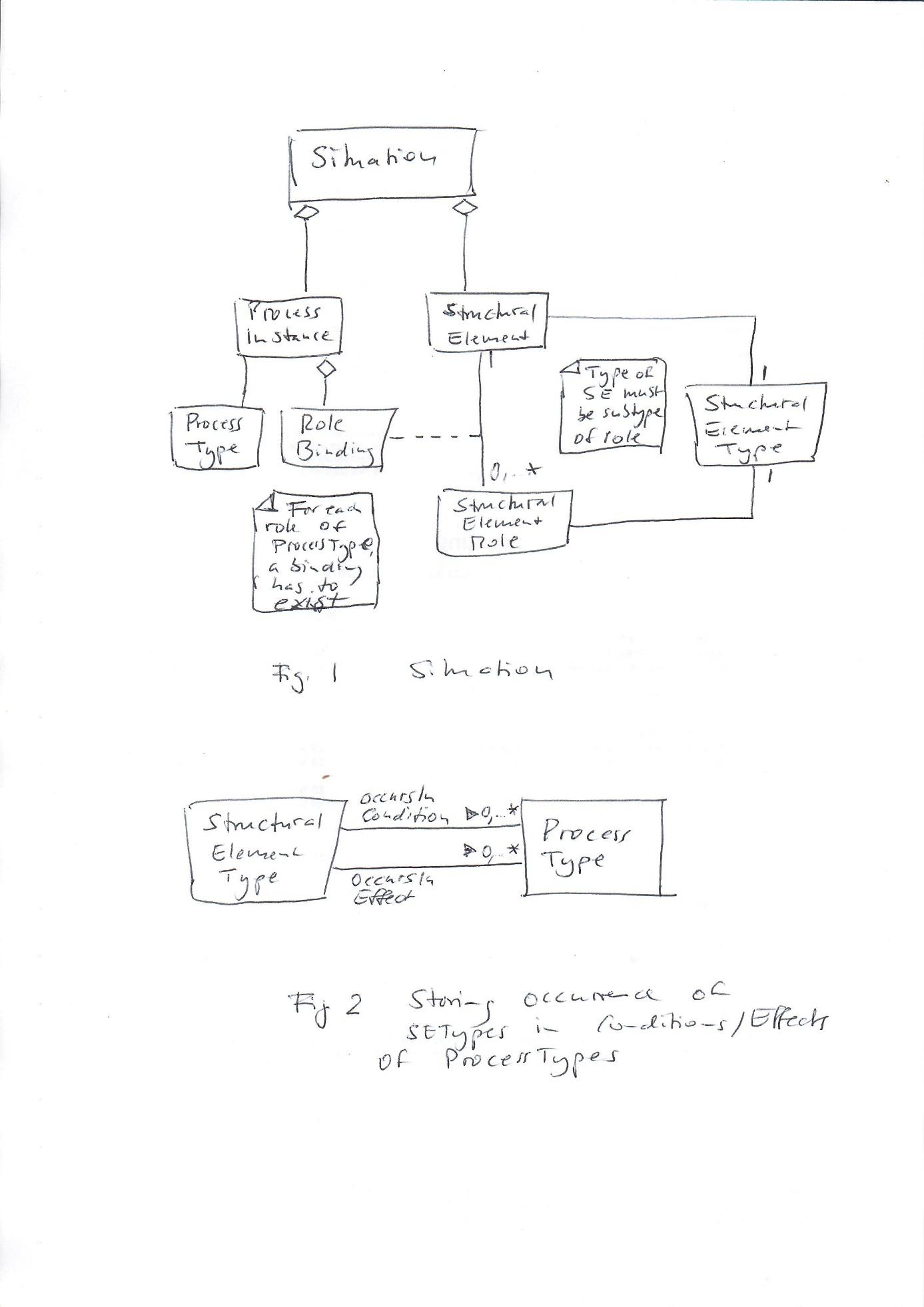
## Basic Idea of the Algorithm

The method ***Library.ForwardCompletionStructural*** will be called iteratively in the forward/backward extension loop and recursively with forward completion. It attempts to avoid unnecessary search and generating redundant process instances.

As a contribution to this objective, the search for ProcessTypes that may be instantiated as part of the forward completion, is limited to those that contain at least one StructuralElementRoles that could be filled by those StructuralElements that have been added in the last (iterative or recursion) step – because, otherwise, they would have been considered and possibly instantiated before.

Secondly, when these selected ProcessTypes are checked for instantiation, the respective method, ***ProcessType.NewInstancesStructural***, has to guarantee that at least one of the bindings of a role in the StructuralConditions refers to a new StructuralElement, because, otherwise, it would create a copy of an existing ProcessInstance.

In order to quickly identify the relevant ProcessTypes, it is suggested to directly access them for each StructuralElementType (separately for conditions and effects), as depicted in Fig. 2. These associations can be created and maintained when ProcessTypes are defined or modified. The representation introduced in section xx to support the generation of rolebindings may serve this purpose, as well.



## Pseudo Code Library.ForwardCompletionStructural

**Input**: *newStructuralElements*: the list of the StructuralElements added by backward completion or by the latest recursive call and *preExistingStructuralElements*: the list of those that existed before. *forwardExtensionStructural* collects the extension of the situation.

**Output**: *forwardExtensionStructural:* the extension of the situation triggered by the additional StructuralElements

**Functionality**: the algorithm retrieves the ProcessTypes whose StructuralConditions contain StructuralElementTypes of the newStructuralElements (or supertypes of them) and checks whether their StructuralConditions can be completely satisfied by (new and old) StucturalElements. If this is the case, the respective ProcessInstances are added to the forward extension, and if their StructuralEffects create new StructuralElements, these become the newStructuralElements, and the method is called recursively.

***Library.ForwardCompletionStructural*** (newStructuralElements,

preExistingStructuralElements,

forwardExtensionStructural)

processTypeCandidates = ∅

FOR newStructuralelement IN newStructuralElements

processTypeCandidates = processTypeCandidates ∪

newStructuralElement.Type.ProcessTypePreconditions

*// The ProcessTypes containing roles satisfied by the Types of the  
 newStructuralElements are collected*

END FOR

newForwardExtensionStructural = ∅

FOR processTypeCandidate IN processTypeCandidates

newForwardExtensionStructural = newForwardExtensionStructural ∪

processTypeCandidate.NewInstancesStructural (newStructuralElements,

preExistingStructuralElements)

*// Collects the instances of ProcessTypes whose Structuralconditions can be*

*satisfied by the existing StructuralElements*

END FOR

*// and adds them to the forwardExtension:*

forwardExtensionStructural = forwardExtensionStructural ∪ newForwardExtensionStructural

newEffectStructuralElements = newForwardExtensionStructural.EffectStructuralElements

/ (newStructuralelements ∪ preExistingStructuralElements)

*// Creates the set of StructuralElements that appear in the StructuralEffects*

*of the latest ForwardExtension and did not exist before*

IF newEffectStructuralElements =/= ∅

*// If there are new StructuralElements as StructuralEffects, then recursive cal:l*

THEN forwardExtensionStructural =

self.ForwardExtensionStructural (newEffectStructuralElements,

newStructuralElements ∪ preExistingStructuralelements,

forwardExtensionStructural)

END IF

RETURN forwardExtensionStructural

**Remark**

The initial call of the method is

*Library.ForwardCompletionStructural* (Observations.StructuralElements, ∅, ∅)

the one after backward completion is

*Library.ForwardCompletionStructural* (BackwardCompletion.newStructuralelements,

preExistingStructuralelements,

∅)

# Computing RoleBindings

## Underlying Ideas

A valid RoleBinding has to

* Be consistent regarding the types: the type of the Object/Relation has to be a subtype of the type of the ObjectRole/RoleRelation in the Binding
* Respect the definition of the RoleRelations: binding a relation to a RoleRelation implies binding the related objects to the ObjectRoles according to the roleRelation.

The approach of the proposed solution is to represent both kinds of restrictions as sets of tables/constraints and combine them to compute using operations of relational algebra/CSP.

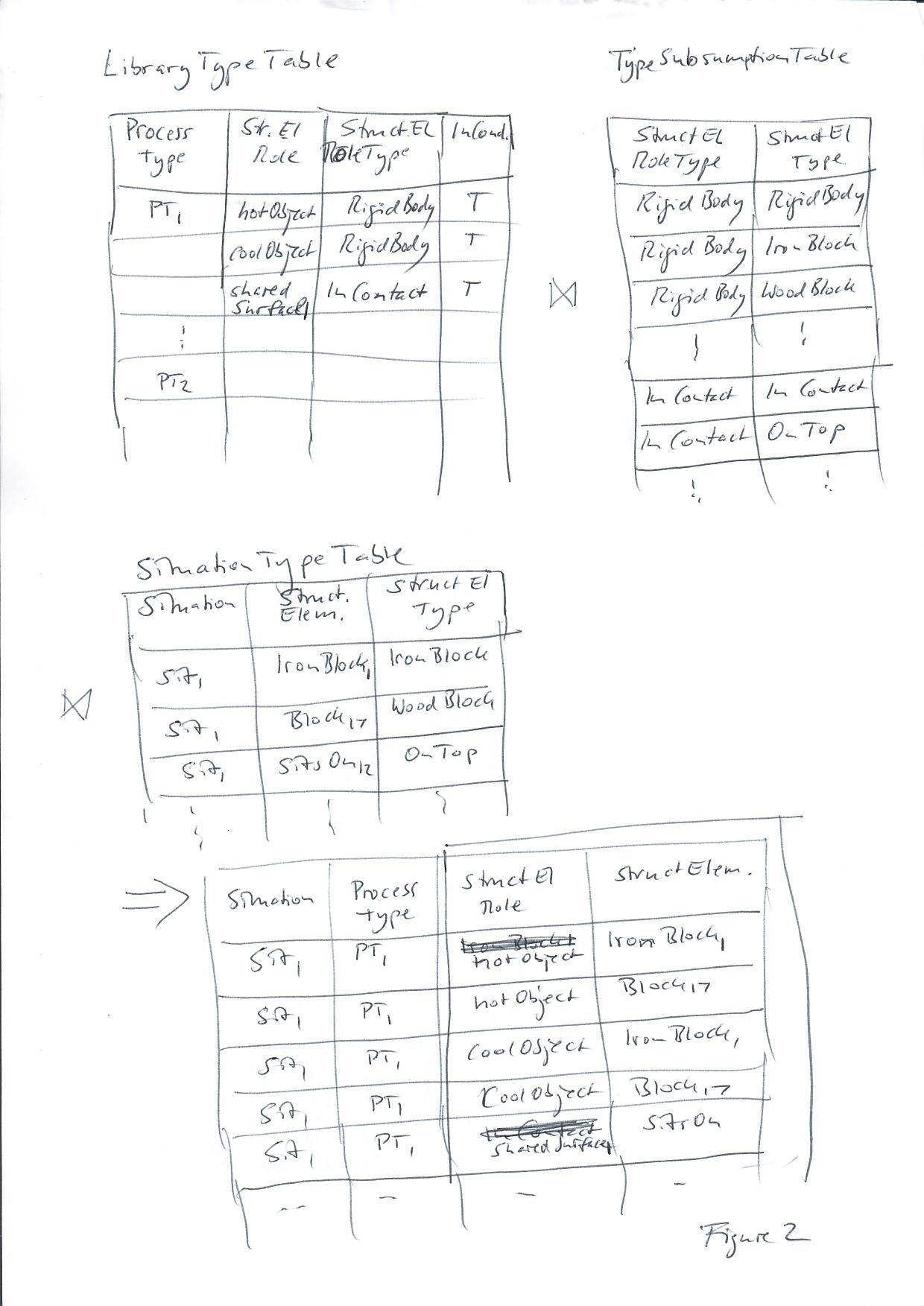
* This could be done incrementally (e.g. by adding the constraints imposed by the bindings of relations one by one), because if adding such a constraint leads to an inconsistency, the procedure can be stopped (there is no proper Rolebinding for a particular RoleRelation, and, hence, the Processtype cannot be instantiated)
* In this case, the constraints produced by RoleRelations should be applied first, because they impose stronger restrictions (on several RoleBindings for ObjectRoles). Perhaps, one should start with Relations that locate objects. There may be domain-specific preferences on the order)
* It is not clear whether such an incremental approach is really required or whether brute-force CSP on all constraints is fast enough.

The following is meant to precisely describe the structure of the required data and the operations, but does not necessarily fix the implementation, for which there may be several ways.

## Tables

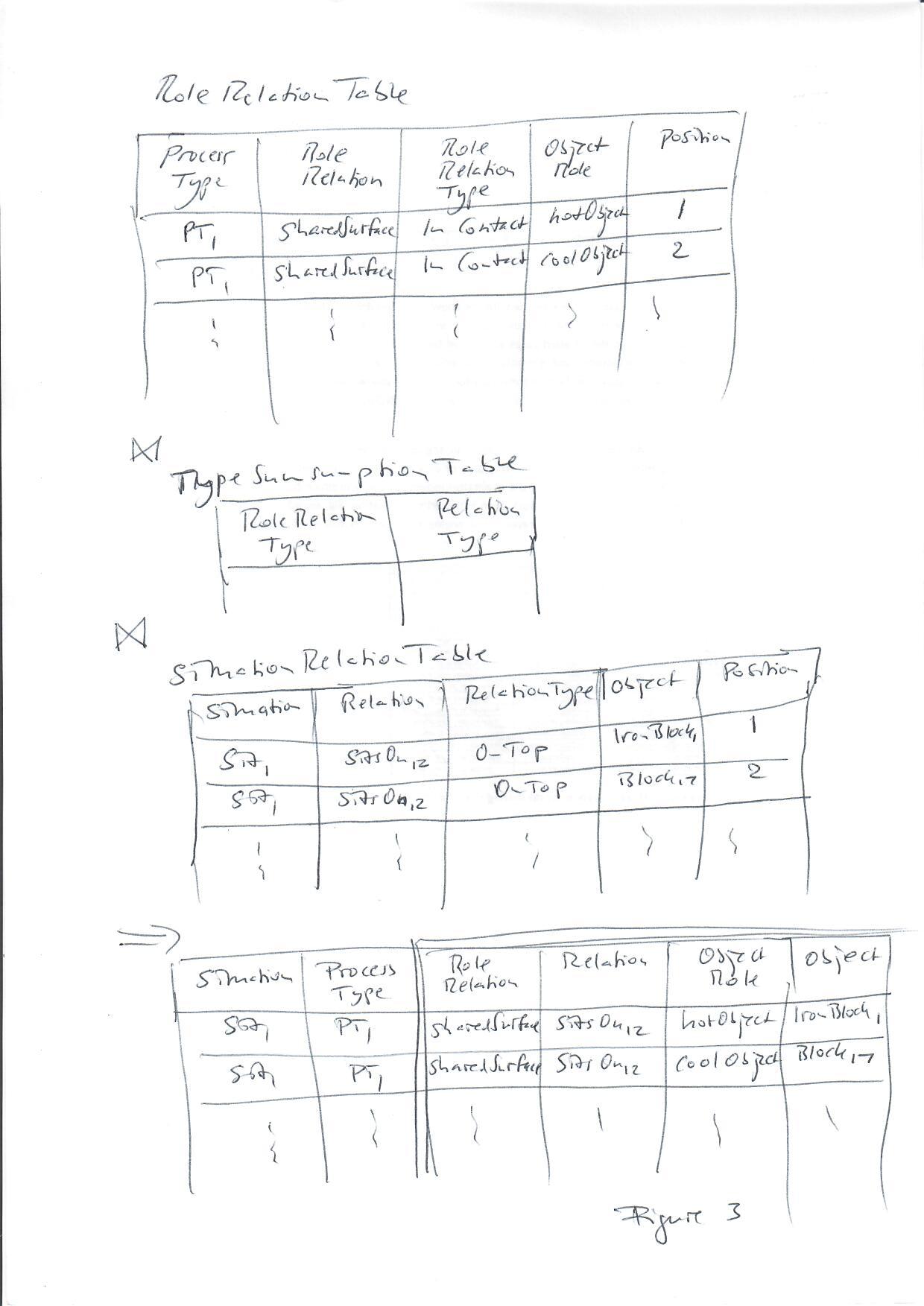
### LibraryTypeTable

In this context, the relevant information to be extracted from the ProcessTypeLibrary concerns the association of StructuralElementTypes to (ObjectRoles/RoleRelations (in the following referred to as StructuralElementTypes. The LibraryTypeTable looks as depicted in Figure 2 (with a column InCondition? Indicating whether the respective StructuralElementRole occurs in the StructuralConditions).



### RoleRelationTable

RoleRelations have a RelationType and fix the position of the related ObjectRoles in its signature. Then Objects and ObjectRoles have to match according to their position (See figure 3).



### TypeSubsumptionTable

This table represents the information from the TypeHierarchy, in collecting all type pairs, where the second one satisfies the first one (Figure 2).

### SituationTypeTable

This table lists all StructuralElements along with their StructuralElementType (Figure 2).

### SituationRelationTable

In analogy to the RolerelationTable, it lists the Relations, their RelationTypes, and the Objects with their position (Figure 3).

## Operations

### Determining Relevant Processes and RoleBindings via Types

The join

LibraryTypeTable ⋈ TypeSubsumtionTable ⋈ SituationTypeTable

contains all ProcessTypes with RoleBindings that satisfy the TypeHierachy (see figure 2). This can be used for several purposes:

* Through Selection Condition=T and projection to ProcessType, it will deliver the ProcessTypes whose StructuralConditions contain StructuralElementTypes that occur in the Situation. i.e. it can be used to implement the method *StructuralElementType.ProcessTypePreconditions* – but perhaps, this is not necessary anymore, because one can use the entire result including the RoleBindings (see next bullet)
* If, for a selected ProcessType, the projection to StructuralElementRole is not a superset of the ObjectRoles/RoleRelations occurring in the StructuralConditions of this ProcessType , then this can be excluded, because its preconditions cannot be satisfied in the situation. Hence, this is stronger than the above criterion.

### RoleBindings via Relations

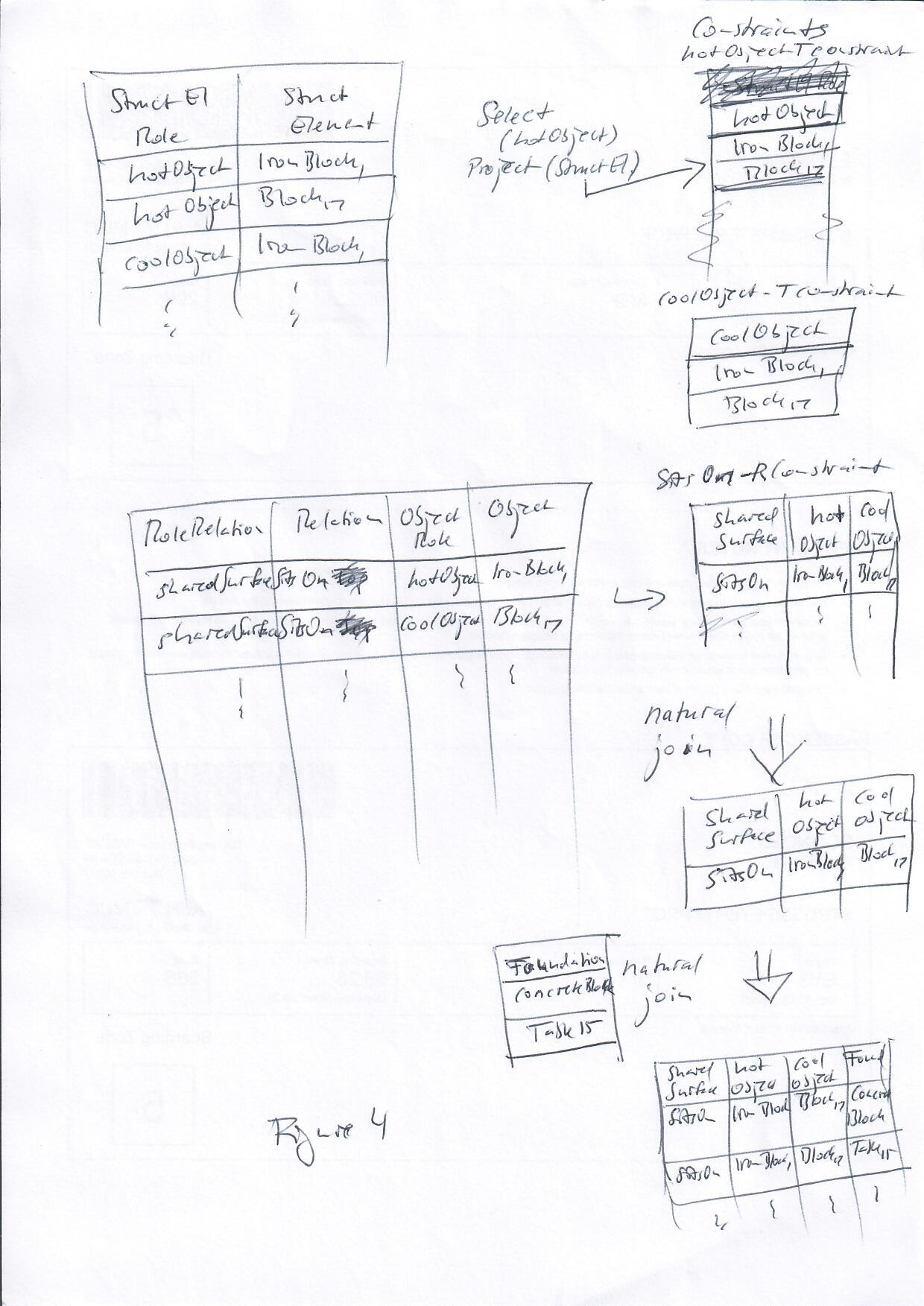
The join

RoleRelationTable ⋈ TypeSubsumtionTable ⋈ SituationRelationTable

produces RoleBindings both for pairs of RoleRelation/Relation and for pairs ObjectRole/Object, where the latter are enforced by the former (Figure 3). Note that, while the Rolebindings regarding relations are correct, the ones for the objects may be incorrect, because the Type of the object may be more general than the one of the ObjectRole and, hence, has to be checked against the type-based RoleBindings of section 3.3.1

### Computing Complete and Consistent RoleBinding

The tables resulting from the two types of operations represent restrictions on binding of the different ObjectRoles/RoleRelations, the type-based ones for individual roles, the relation-based ones for combinations of RoleRelations and ObjectRoles (illustrated in Figure 4). These restrictions altogether determine the possible RoleBindings. The idea is to turn them into tables/constraints that can then be used to construct the RoleBindings by relational algebra operations/CSP (natural join or constraint solving). In this transformation, the StructuralElementRoles would become variables and their common domain would be the set of all Structuralelements.



There could be different ways of doing this:

* Constructing the type-based table including all StructuralElementRoles and then the join with all relation-based tables.
* Iterative expansion of the RoleBindingTable/constraint network by starting with the (strongest) relations and adding further constraints step by step. If an inconsistency is detected early on, other constraints can be ignored. A reasonable heuristic could be to apply, after a relation-based constraint, the type-based constraints for the ObjectRoles occurring in it.

Note that even though only ProcessTypes may have been selected that contain StructuralElementTypes of the latest objects, the algorithm would also generate RoleBindings involving only old elements, which have to be eliminated.

## StructuralElementType.ProcessTypePreconditions

**Input**: *-*

**Output**: The list of all ProcessTypes that contain a role of the StructuralelementType or a supertype of it.

**Functionality**: Accesses the OccursIncondition association and recursively searches the supertypes

*StructuralElementType.ProcessTypePreconditions* ()

xxxxxxxxxxxxxxx

## ProcessType.NewInstancesStructural

**Input**: *newStructuralElements*: the most recently added StructuralElements; *preExistingStructuralElements*: the previous ones.

**Output**: The list of all ProcessInstances, each with a complete RoleBinding, that contain at least one of the newStructuralElements in a binding for one role from the Structuralconditions.

**Functionality**: This is the core method for matching precondtions with existing StructuralElements and instantiating them, which potentially includes the creation of new StructuralElements as effects.

***ProcessType.NewInstancesStructural*** (newStructuralElements,

preExistingStructuralElements)

xxxxxxxxxxxxxxxxxxxx

## Situation.EffectStructuralElements

**Input**: -

**Output**: The list of all StructuralElements occurring as effects in one of the ProcessInstances of the situation.

**Functionality**: Iterate over the ProcessInstances and collect the StructuralElements that occur in RoleBindings for StrcuturalEffects.

Situation.EffectStructuralElements ()

Xxxxxxxxxxxxxx

**Conjecture**

If there were ProcessInstances in this set that are inconsistent with each other, e.g. if they bind the same object to different locations, this would indicate a bug in the Ontology (normally ProcessType definition). This conjecture is based on the consideration that Processtypes specify deterministc processes. Question: Could there be initial situations (i.e. the observations) that are consistent in themselves, but trigger processes that may produce such inconsistencies?