Refactoring Object-Oriented Specifications with Data and Processes*

Thomas Ruhroth and Heike Wehrheim

Department of Computer Science
University of Paderborn
33098 Paderborn, Germany
{ruhroth,wehrheim}@uni-paderborn.de

Abstract. Refactoring is a method for improving the structure of programs/specifications as to enhance readability, modularity and reusability. Refactorings are required to be *behaviour-preserving* in that – to an external observer – no difference between the program before and after refactoring is visible. In this paper, we develop refactorings for an object-oriented specification formalism combining a state-based language (Object-Z) with a process algebra (CSP). In contrast to OO-programming languages, refactorings moving methods or attributes up and down the class hierarchy, in addition, need to change *CSP processes*. We formally prove behaviour preservation with respect to the failures-divergences model of CSP.

1 Introduction

Refactoring is a technique which has long been used by programmers to improve the structure of their code once it got unreadable. The word "refactoring" as a general term for frequently occurring clean-up operations on programs has been coined by Fowler [Fow04]. The book [Fow04] collects a large number of refactorings operating on different levels: the level of methods only, those of classes and of the class hierarchy. As Fowler puts it, all these refactorings "should not change the externally visible behaviour of a program". For programs, this type of behaviour preservation is checked via *testing*: there are a number of tests associated with every (part of a) program which are being run before and after the refactoring. An application of a particular refactoring thus does not a priori guarantee behaviour preservation but has to be tested.

This is different for refactorings for formal specifications: the formal semantics allows for a *proof of correctness* of a refactoring, and thus ensures behaviour preservation. Thus, while refactorings for OO-programs are usually stated via an example, refactorings for formal specifications are given by pairs of templates describing before and after state of a refactoring. These template pairs are proven to guarantee behaviour preservation with respect to the formal semantics of

* This work was partially funded by the German Research Council DFG under grant WE 2290/6-1.

the specification language. Thus, whenever some parts of a specification are an instantiation of a before template, they can be replaced with the proper instantiation of the corresponding after template. Additional constraints might constrain the application of the pattern. An overview of these kind of formal approaches to refactoring can be found in [MT04], in particular [MS04] and [MS06] follow this approach for Object-Z specifications, one of the formalisms we will be interested in here. While a lot of the approaches surveyed in [MT04] show behaviour-preservation only for specific properties (e.g. certain invariants of classes or relationships between objects), the basis for the correctness proof of [MS04] is the notion of *data refinement* [dE98,DB01] in Object-Z. Refinement exactly guarantees the intended substitutability requirement: for an external observer the classes before and after refactoring are not different, and this holds for any kind of (external) observation on the class.

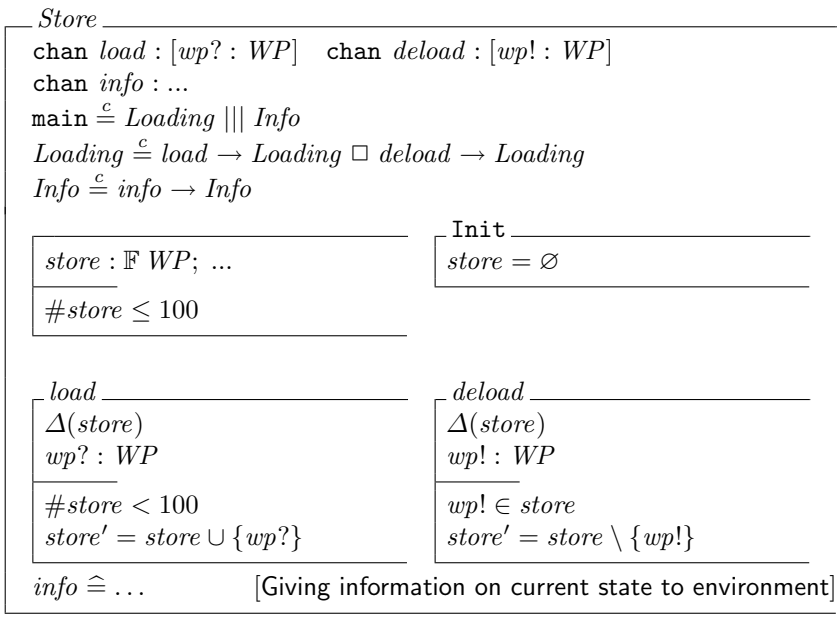
In this paper, we study refactorings for a formal specification language which in addition to state-based descriptions in Object-Z [Smi00] allows for a description of the *dynamic behaviour* via the process algebra CSP [Hoa85,Ros97]. This combination, called CSP-OZ [Fis97], has a semantics defined in terms of the failures-divergences model of CSP. The integration of two orthogonal formalisms gives us a convenient way of modelling both data, methods and the ordering of method executions. For the refactorings, this additional view in our specifications however imposes additional complexity. A change on the Object-Z side most often requires a corresponding change in the process. This in particular applies to refactorings on the level of the class hierarchy where the movement of a method up or down the hierarchy may involve a corresponding move of CSP process parts up or down classes.

As our notion of correctness of refactorings, we use refinement as well as it guarantees the required behaviour preservation. In the combination CSP-OZ, the appropriate notion of refinement is however *process refinement* (failures-divergences refinement), coming from the CSP semantics. Refactorings are only correct if they preserve the failures-divergences semantics of all involved classes, up to refinement. We aim at defining generally usable *templates* for refactorings such that correctness is guaranteed for every concrete instantiation. Here, we present a general proof strategy for CSP-OZ refactorings based on an expansion into CSP_Z given in [Fis00], which in turn is based on a similar semantics for Object-Z [Smi00]. This proves to be a convenient approach since (most of the) refactorings can thus be shown to be correct by syntactical rewritings of schemas only. We, however, also present a correctness proof for a refactoring which involves an explicit construction of a refinement. The whole approach is exemplified with a CSP-OZ specification in which we refactor single classes as well as introduce a class hierarchy via refactoring.

2 Background

We start with a first part of our case study by which we introduce the formalism CSP-OZ, its semantics and the notion of refinement. The following, only partially

given class specification is describing one part of a manufacturing system, namely a store. Stores are holding workpieces which can be loaded/deloaded from and to autonomous transportation agents. For this, we first of all need two basic types for workpieces and for names of transportation agents (Hts): $[WP, Hts]$. The class *Store* is a CSP-OZ class consisting of a CSP part describing the dynamic behaviour (ordering of operations) of the class, and an Object-Z part describing the static behaviour (data and operations). Parts not relevant for our refactorings are being omitted (written as ...).



The specification consists of a declaration of the *interface* of the class as a number of channels for communication with other classes (viz. objects). Here, channels *load*, *deload* and *info* are declared together with their signatures. After this, a CSP process `main` is given defining the dynamic behaviour of the class (viz. its objects). For class *Store* this is an interleaving ($|||$) of the processes *Loading* and *Info*. Process *Info* just repeatedly executes operation *info* ($\rightarrow$ is the prefix operator describing sequencing), and process *Loading* consists of an external choice ($\Box$) over either a *load* or a *deload*.

The remaining part of the specification defines the variables (sometimes also called fields) in the state schema (a variable *store* with an invariant fixing the size of the store), the initial values (in the init schema) and the operations. An operation schema typically consists of a Δ-list, declaring the variables which are allowed to be changed, and input and output variables (denoted by ? and !, respectively) together with a predicate defining constraints on state changes. Here, primed variables denote variables in the after state. For instance, operation *load* is allowed to change variable *store*, has an input variable *wp*? and a predicate stating the precondition of the operation (*store* has not to be full) and the

outcome (the workpiece in the input is added to the store). In addition, CSP-OZ allows to specify *inheritance* relationships between classes (not present here), denoted by `inherit` *superclassname*.

Semantics. This combination of CSP and Object-Z has a well-defined semantics in terms of the failures-divergences model of CSP [Fis00]. The semantics is defined by first translating a CSP-OZ specification to $\mathrm{CSP_Z}$ (a CSP dialect with Z syntax for expressions and declarations), from which the failures and divergences are then derived. For proving the correctness of our refactorings we only need to go to the level of $\mathrm{CSP_Z}$, thus we will only explain this part. The basic idea is to model the CSP part and the Object-Z part of the CSP-OZ specification in $\mathrm{CSP_Z}$. These two parts can then be combined into a semantics of the whole CSP-OZ specification using the parallel composition operator $({}_A\|_B)$ of CSP. More specifically, the semantics of a CSP-OZ class C is

$$proc(C) = procC(C) {}_{Chans(procC(C))}\|_{Chans(procZ(C))} procZ(C),$$

where $procC(C)$ is the semantics of the CSP part and $procZ(C)$ those of the Object-Z part. The function *Chans* computes the channels used in a process expression, and ${}_A\|_B$, A, B set of events, is the parallel composition allowing the left process to communicate on events in A and the right in B with synchronisation on events in the intersection. Thus CSP and Object-Z part synchronise on joint operations. The semantics $procC$ is either simply the CSP process `main` (if the class has no superclass), or the parallel composition of `main` with the main process of the superclass S, again synchronising on common operations:

$$procC(C) = \texttt{main}_{Chans(\texttt{main})}\|_{Chans(procC(S))} procC(S)$$

The semantics of the Object-Z part ($procZ$) is defined by first mapping Object-Z constructs to Z and then transforming them to $\mathrm{CSP_Z}$. In this paper we will in particular use the functions **init**() and **state**(), which map Object-Z constructs to pure Z schemas. The function **init**() gives a Z schema representing the initialisation, and **state**() a Z schema representing the state of the class. These and some other functions are used within $procZ$. Due to lack of space, we omit these definitions here, for details and rules see [Smi00,Fis00].

Refinement. Correctness, viz. behaviour preservation of refactorings, is in our setting defined via *refinement* [dE98,DB01]. Refinement guarantees *substitutability*: while internal representations may change, the changes should not be externally visible. Since refactorings are usually applicable in both directions (a method pushed up to a superclass or down to the subclasses), we need refinement in both directions.

For the specification formalism CSP-OZ, two notions of refinement are of importance: *data refinement* from Object-Z and *failures-divergences refinement* from CSP. We start with the former. Data refinement is defined as substitutability of one specification by another, and usually proven by forward and backward

simulations. Here, we just need forward simulations and thus give this definition only. It assumes to have two Object-Z classes A and C given (or the Object-Z parts of two CSP-OZ classes), which both consist of a state schema, an initialisation schema and some operation schemas: $A = (AState, AInit, \{AOp_i\}_{i \in I})$ and $C = (CState, CInit, \{COp_i\}_{i \in I})$, where I is some index set for operations.

Definition 1. *C is a* forward simulation *of A, $A \sqsubseteq_D C$, if there is a retrieve relation R between $AState$ and $CState$ such that the following hold:*

1. *Initialisation:* $\forall\, CState \bullet CInit \Rightarrow (\exists\, AState \bullet AInit \wedge R)$,
2. *Applicability:* $\forall\, i \in I, \forall\, AState, CState \bullet R \Rightarrow (preAOp_i \Leftrightarrow preCOp_i)$,
3. *Correctness:* $\forall\, i \in I, \forall\, AState, CState, CState' \bullet$
$$R \wedge COp_i \Rightarrow \exists\, AState' \bullet R' \wedge AOp_i.$$

Basically, the idea is to find a relation between the variables in A and C such that the operations in C are applicable in a state if and only those in A are applicable in a related state, and the execution of an operation in C can correspondingly be carried out in A leading to related states again. While the definition of variables and operations may have been changed in C, its externally visible behaviour cannot be distinguished from A.

Data refinement, or more specifically forward simulation, is used when we need to look at the Object-Z part of a CSP-OZ specification in isolation. In the combination, the basis for a definition of refinement is the semantics for CSP-OZ, i.e. the failures-divergences model of CSP. Again, we will not actually compute failures and divergences of processes, but work on the level of CSP processes only. For CSP processes P and Q, we write $P \sqsubseteq_{FD} Q$ if Q is a process (failures-divergences) refinement of P.

Finally, we need to know the relationship between these two kinds of refinement. A lot of research has recently been carried out on the comparison of data and process refinement, the relevant result here is the following (from [Fis00]).

Theorem 1. *Let A, C be Object-Z parts of CSP-OZ classes. Then*

$$A \sqsubseteq_D C \Rightarrow procZ(A) \sqsubseteq_{FD} procZ(C)\ .$$

Furthermore, process refinement is preserved under parallel composition ([Ros97]), which is the operator used for combining the processes of CSP and Object-Z part.

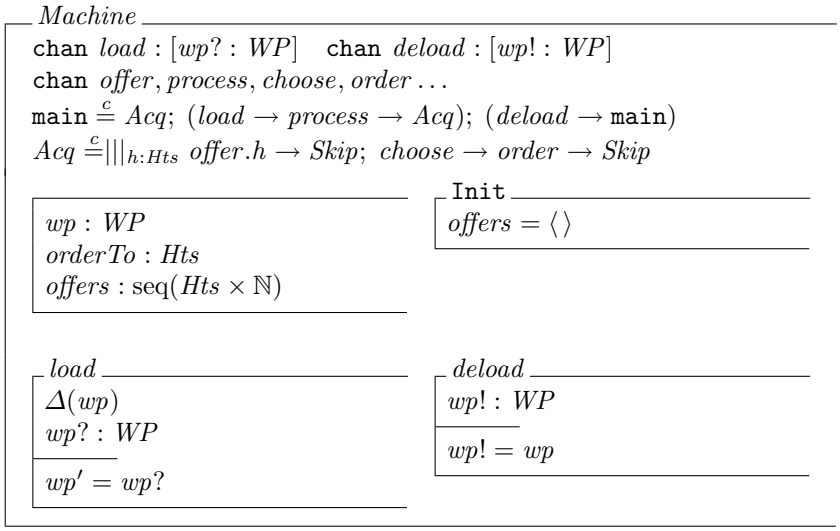
Theorem 2. *Let P_1, P_2, Q_1, Q_2 be CSP processes, A, B sets of events. Then*

$$P_1 \sqsubseteq_{FD} Q_1 \wedge P_2 \sqsubseteq_{FD} Q_2 \Rightarrow P_1 {}_A\|_B P_2 \sqsubseteq_{FD} Q_1 {}_A\|_B Q_2\ .$$

As a consequence, we can separately show a data refinement relationship on the Object-Z parts and a process refinement on the CSP parts, and obtain a process refinement for the combination. Thus refactorings operating on the Object-Z part alone can be proven correct without having to look into the CSP part.

3 Case study

Next, we continue our example and extend it with another class. These two classes are then the starting point for our refactorings. The second class specifies machines in the manufacturing system. Similar to stores, machines can load and deload workpieces. The machine is furthermore an active entity as it actively seeks to find some transportation agent for a job. The CSP process *Acq* below describes the events carried out for an acquisition of a transportation agent (essentially getting offers from agents, choosing the offer with the smallest cost and ordering this agent), their exact meaning is however not relevant for our aims. In between these operations, loading, processing and deloading of workpieces takes place. The operator ; denotes sequential composition. The variables *orderTo* and *offers* are used for the acquisition of transporation agents.



There are some obvious similarities between this class and class *Store*: both store workpieces (*Store* up to a hundred, *Machine* only one) and both load and deload workpieces. We could thus think of having a common superclass for both classes describing these common functionalities. This would result in a specification which does not duplicate the description of two operations, and there would be a single point in the specification in which changes to these operations have to be made (for instance during a refinement to code). Our objective is thus now to introduce a common superclass to *Store* and *Machine*, and move common variables and operations to this superclass. This goal is in the following achieved through a number of successive refactorings.

First refactoring. Looking at the two definitions of operations *load* and *deload*, which are candidates for operations of the superclass, we see that they are different. Our first refactoring thus works towards making them similar. In *Machine*

we change state, init schema and operations to

$$store : \mathbb{F}\ WP$$
$$orderTo : Hts$$
$$offers : \mathrm{seq}(Hts \times \mathbb{N})$$
$$\#store \leq 1$$

Init

$$store = \varnothing$$
$$offers = \langle\,\rangle$$

load

$$\Delta(store)$$
$$wp? : WP$$
$$\#store < 1$$
$$store' = store \cup \{wp?\}$$

deload

$$\Delta(store)$$
$$wp! : WP$$
$$wp! \in store$$
$$store' = store \setminus \{wp!\}$$

Instead of having a variable of type *WP* we now have a set of *WP*s of size one. This looks like a data refinement on the Object-Z side but it is not. The preconditions of *load* and *deload* in the Object-Z part are strengthened: while previously both operations were always enabled, they are now only enabled when *store* is currently empty or filled, respectively. Due to the blocking semantics of Object-Z such a change becomes visible to an observer: operation *load* might sometimes be disabled. Fortunately, in connection with the CSP part it is a correct refinement since the CSP part ensures an alternating execution of *load* and *deload*, thus the blocking has already been present in the previous specification of *Machine*. The correctness of this transformation, i.e. behaviour preservation, can be proven using a technique presented in [DW06]. We thus will not further look at the correctness of this refactoring.

Second refactoring. Our next refactoring tackles the remaining difference between *Store* and *Machine* as far as the field *store* is concerned. We carry out the refactoring "Replace Magic Number with Symbolic Constant" [Fow04] in both classes, replacing the numbers 100 and 1 by a variable *capacity* which is then initially set to the respective value. The relevant part of *Store* looks like this (similarly for *Machine*):

Store

$$store : \mathbb{F}\ WP$$
$$capacity : \mathbb{N}$$
$$\#store \leq capacity$$

$\ldots$

Init

$\ldots$
$$capacity = 100$$

[plus replacing the number in *load*]

Third Refactoring. Looking at *Store* and *Machine* we now see that they share similar variables and methods. Hence a superclass can be extracted from them,

and variables, part of the initialisation and methods pulled upwards to this class. The next refactoring (called "Extract Superclass") is a combination of four smaller refactorings (all from [Fow04]): "Extract Class" creates an empty class (*Station*) and makes *Store* and *Machine* inherit from this class, "Pull Up Field" moves variables *store* and *capacity* from subclasses to superclass, "Pull Up Init" moves initialisation of *store* to superclass (but not of *capacity* since this is different in the two subclasses) and finally "Pull Up Method" moves methods *load* and *deload* up to *Station* (shown next, omitting interface declaration).

Station

$$store : \mathbb{F}\ WP$$
$$capacity : \mathbb{N}$$
$$\#store \leq capacity$$

`Init`

$$store = \varnothing$$

load

$$\Delta(store)$$
$$wp? : WP$$
$$\#store < capacity$$
$$store' = store \cup \{wp?\}$$

deload

$$\Delta(store)$$
$$wp! : WP$$
$$wp! \in store$$
$$store' = store \setminus \{wp!\}$$

Both *Store* and *Machine* inherit from *Station*, i.e.

Store

`inherit` *Station*
...

Machine

`inherit` *Station*
...

and both do not contain definitions of *load*, *deload*, *store* and *capacity* anymore (being inherited from *Station*), only the initialisation of *capacity* remains in the subclasses as it differs in the two classes.

Fourth Refactoring. Last, we have to look at the CSP part. The two classes have quite different CSP parts, in particular both also refer to operations other than *load* and *deload*. Thus neither the CSP part of *Store* nor that of *Machine* can be completely moved to the superclass. However, one part of *Store* could potentially be moved to *Station*, namely we could define the CSP part in *Station* as

$$\begin{aligned} &\texttt{main} \stackrel{c}{=} Loading \\ &Loading \stackrel{c}{=} load \rightarrow Loading \\ &\qquad\qquad \Box\ deload \rightarrow Loading\ , \end{aligned}$$

and change the CSP part of *Store* to $\texttt{main} \stackrel{c}{=} Info$. This refactoring is called "Pull up CSP" (not from [Fow04]); it is moving one part of a parallel composition in a CSP process of a subclass to a superclass. However, due the semantics of

inheritance (parallel composition of CSP parts of sub- and superclass) this affects the CSP part of *Machine* as well. We have to make sure that the CSP process obtained by this parallel composition is equivalent to the old CSP process. To this end, we first rephrase the CSP part of *Machine* (refactoring "Rephrase CSP", not from [Fow04]) to a form where this parallel composition is explicitly visible and show behaviour preservation for this transformation. In *Machine* we get

$$\begin{aligned}
&\texttt{main} \stackrel{c}{=} Loading \,{}_{Chans(Loading)}\|_{Chans(Work)}\, Work\\
&Loading \stackrel{c}{=} load \rightarrow Loading \,\Box\, deload \rightarrow Loading\\
&Work \stackrel{c}{=} Acq;\ (load \rightarrow process \rightarrow Acq);\ (deload \rightarrow Work)\\
&Acq \stackrel{c}{=} |||_{h:Hts}\ offer.h \rightarrow Skip;\ choose \rightarrow order \rightarrow Skip
\end{aligned}$$

Equivalence, i.e. refinement in both directions, between this new and the old process of *Machine* can be automatically shown using the CSP model checker FDR [FDR97]. Then, *Loading* can be moved upwards to superclass *Station* from both *Store* and *Machine* preserving the overall semantics.

4 Correctness of refactorings

In the example above we have seen several different refactorings, affecting only the CSP part, only the Object part or both parts.

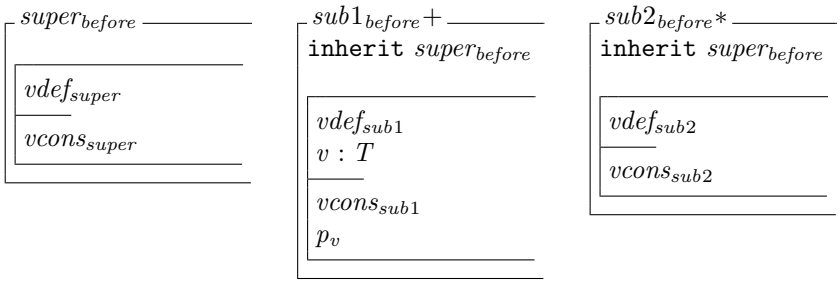
Object-Z. Refactorings which only affect the classes being changed are called *inner refactorings*. Such inner refactorings of the Object-Z part can be easily derived from the inner refactorings of Object-Z itself (using an approach presented in [Ruh06]), and can - due to Theorem 1 - proven correct by looking at the Object-Z part in isolation. Four refactorings of the example fulfil this condition: "Pull Up Field", "Pull Up Method", "Pull Up Init" and "Replace Magic Number with Symbolic Constant". Here we just prove correctness of "Pull Up Field", the other proofs are similar.

All of our refactorings will be formally described by a template consisting of three parts: A (possibly empty) *condition* stating application conditions for the refactoring, and two patterns of specifications stating the *before* and *after* state of the refactoring. In the patterns we will not have concrete variables, but metavariables which can be instantiated in an arbitrary way. The template for "Pull Up Field" describes how and when a variable v can be moved from (one or more) subclasses to a superclass.

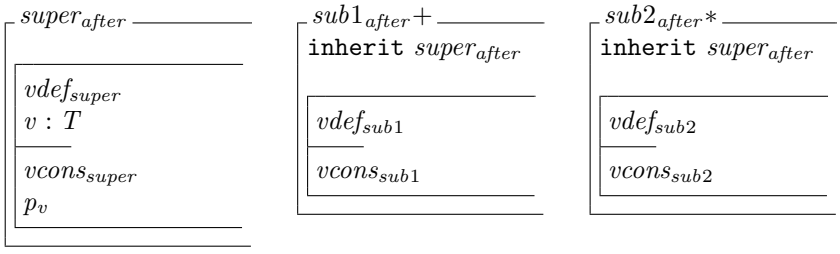
CONDITION:

$$\begin{aligned}
&v \notin \mathbf{vars}(\mathbf{state}(super_{before})) \wedge v \notin \mathbf{vars}(\mathbf{state}(sub2_{before}))\\
&\exists\, v : T \bullet p_v\\
&vars(p_v) \subset \{v\} \cup \mathbf{vars}(\mathbf{state}(super_{before}))
\end{aligned}$$

BEFORE:



AFTER:



This refactoring assumes that there is a superclass $super_{before}$ with at least one subclass of type $sub1_{before}$ (denoted by $+$, regular expression) and zero, one or more subclasses of type $sub2_{before}$ (denoted by $*$). The subclasses of type $sub1_{before}$ all have a field v with the same type T and a predicate p_v constraining the values of v. In addition they may have other (differing) fields (summarised in $vdef_{sub1}$) with predicates $vcons_{sub1}$ over them. Note that the predicate $vcons_{sub1}$ may also constrain variable v. Subclasses of type $sub2_{before}$ and the superclass all do not have the variable v in their state schema. Furthermore, the condition requires that there is at least one possible value for v such that the predicate p_v is fulfilled. The after template describes the specification after the refactoring: field v and its predicate p_v have been pulled upwards into the superclass. Note that when applying this refactoring to our example, we first pull up one variable (e.g. *store*) and an empty predicate (*true*), and in the second step the other variable (here *capacity*) and the predicate $\#store \leq capacity$.

For correctness, we need to prove that the superclass and all subclasses remain equivalent (wrt. refinement) under this transformation. We do this in three steps: first, we show that the classes which previously have included the variable remain the same, second, we prove the same for the classes which have not previously included the variable (i.e. $sub2$), and third, we have to prove equivalence for the superclass. We start with proving equivalence for a class of type $sub1$. The important part is to prove that the semantics of the state does not change, i.e. $\mathbf{state}(sub1_{before}) = \mathbf{state}(sub1_{after})$. Using the semantics rules from

the language definition (Chapter 4) of [Smi00] we can transform the left part of the equation:

$$
\begin{aligned}
&\mathbf{state}(sub1_{before}) \\
&= [self : sub1_{before}] \wedge (\mathbf{state}(super_{before})/(self)) \\
&\quad \bullet \mathbf{state}([v : T;\ vdef_{sub1} \mid vcons_{sub1};\ p_v]) \\
&= [self : sub1_{before}] \wedge (\mathbf{state}(super_{before})/(self)) \\
&\quad \bullet \mathbf{state}([v : T \mid p_v]) \bullet \mathbf{state}([vdef_{sub1} \mid vcons_{sub1}]) \\
&= [self : sub1_{before}] \wedge ((\mathbf{state}(super_{before}) \wedge \mathbf{state}([v : T \mid p_v]))/(self)) \\
&\quad \bullet \mathbf{state}([vdef_{sub1} \mid vcons_{sub1}) \\
&= [self : sub1_{before}] \wedge \mathbf{state}([vdef_{super};\ v : T \mid vcons_{super};\ p_v])/(self)) \\
&\quad \bullet \mathbf{state}([vdef_{sub1} \mid vcons_{sub1}) \\
&= [self : sub1_{after}] \wedge (\mathbf{state}(sub1_{after})/(self)) \\
&\quad \bullet \mathbf{state}([vdef_{sub1} \mid vcons_{sub1}) \\
&= \mathbf{state}(sub1_{after})
\end{aligned}
$$

Essentially, the state of the class, which owns the variable v before applying the refactoring does not change through this refactoring. From this we conclude that the class before and after the refactoring are equivalent (under data refinement).

Next, we prove that classes which did not include the variable, are equivalent before and after applying the refactoring. This is more complicated than the first part because we have to show that the enhanced state does not change the behaviour. Here we have to use another proof technique because the class does not remain equivalent as far as its state is concerned. It is to be proven that the class before and after applying the refactoring are refinements of each other using forward simulation. Here, we only prove that $sub2_{before}$ is a refinement of $sub2_{after}$. We have to show that there is a schema R, which fulfils the conditions of Definition 1. The state of the refactored class is the old state combined with the variable v and some predicates p_v:

$$\mathbf{state}(sub2_{before}) \wedge [v : t \mid p_v] \equiv \mathbf{state}(sub2_{after})$$

For this, we choose R to be the identity on the variables of $sub2_{before}$. We immediately get $\mathbf{init}(sub2_{before}) = \mathbf{init}(sub2_{after})$ and $sub2_{before}.Op_i = sub2_{after}.Op_i$, because the definition is not modified and the variable v is not used in any of them. We begin with the initialisation condition from Definition 1:

$$
\begin{aligned}
&\forall \mathbf{state}(sub2_{before}) \bullet \mathbf{init}(sub2_{before}) \Rightarrow \exists(\mathbf{state}(sub2_{after}) \bullet \mathbf{init}(sub2_{after}) \wedge R) \\
\equiv\ & \\
&\forall \mathbf{state}(sub2_{before}) \bullet \mathbf{init}(sub2_{before}) \\
&\quad \Rightarrow \exists(\mathbf{state}(sub2_{before}) \wedge [v : T \mid p_v] \bullet \mathbf{init}(sub2_{before}) \wedge R) \\
\equiv\ & \quad \{sub2_{before} \text{ and } \mathbf{init}(sub2_{before}) \text{ do not use } v\} \\
&\forall \mathbf{state}(sub2_{before}) \bullet \mathbf{init}(sub2_{before}) \\
&\quad \Rightarrow (\mathbf{state}(sub2_{before}) \wedge \mathbf{init}(sub2_{before}) \wedge \exists\, v : T \mid p_v \bullet R) \\
\equiv\ & \quad \{ \text{ Definition of } R\} \\
&\forall \mathbf{state}(sub2_{before}) \bullet \mathbf{init}(sub2_{before}) \\
&\quad \Rightarrow (\mathbf{state}(sub2_{before}) \wedge \mathbf{init}(sub2_{before}) \wedge \exists\, v : T \bullet p_v)
\end{aligned}
$$

$\equiv \exists\, v : T \bullet p_v$
$\equiv$ { Assumption }
$\quad true$

We omit the simple proof of applicability and go straight to the proof of the correctness condition of forward simulation:

$\forall \mathbf{state}(sub2_{after}), \mathbf{state}(sub2_{before}), \mathbf{state}(sub2_{before})' \bullet R \wedge sub2_{before}.Op_i$
$\quad \Rightarrow \exists \mathbf{state}(sub2_{after})' \bullet R' \wedge sub2_{after}.Op_i$
$\equiv$
$\forall \mathbf{state}(sub2_{before} \wedge [v : T \mid p_v]), \mathbf{state}(sub2_{before}), \mathbf{state}(sub2_{before})'$
$\bullet R \wedge sub2_{before}.Op_i \Rightarrow \exists \mathbf{state}(sub2_{before} \wedge [v : T \mid p_v])' \bullet R' \wedge sub2_{after}.Op_i$
$\equiv$
$\forall \mathbf{state}(sub2_{before} \wedge [v : T \mid p_v]), \mathbf{state}(sub2_{before}), \mathbf{state}(sub2_{before})'$
$\bullet R \wedge sub2_{before}.Op_i \Rightarrow \exists \mathbf{state}(sub2_{before} \wedge [v : t \mid p])' \bullet R' \wedge sub2_{before}.Op_i$
$\equiv$ { Definition of R and v is not in Δ}
$\forall [v : T \mid p_v], \mathbf{state}(sub2_{before}), \mathbf{state}(sub2_{before})' \bullet R \wedge sub2_{before}.Op_i$
$\Rightarrow \exists [v : t \mid p]' \bullet v = v'$
$\equiv$ { Assumption }
$\forall [v : T \mid p_v], \mathbf{state}(sub2_{before}), \mathbf{state}(sub2_{before})' \bullet R \wedge sub2_{before}.Op_i$
$\Rightarrow true$
$\equiv true$

Thus we have proven that the class $sub2_{before}$ is a refinement of $sub2_{after}$. The proof for the common superclass is analogous to the proof of $sub2$. In a similar way we can prove correctness of the other inner refactorings on the Object-Z part, e.g. "Replace magic number with Symbolic Constant", "Pull Up Method" and "Pull Up Init".

CSP. Next we will look at a refactoring only changing the CSP part of a class. This kind of refactoring is used here in two ways. First, we may want to transform the CSP part to an equivalent one within the CSP-OZ class ("Rephrase CSP"). We use "Rephrase CSP" to bring the CSP part into a shape, in which we can apply the second CSP refactoring, namely "Pull Up CSP". Both can be proven by concentrating on the CSP part alone (Theorem 1). Hence, we can simply prove that the CSP part before and after applying the refactoring is the same.

For "Rephrase CSP" there are two possibilities: we have to show that the CSP-part before and after refactoring is equivalent wrt. the failures-divergences semantics of CSP, and this can either be done by using some of the equivalence rules of CSP (see e.g. [Ros97]) or explicitly asking the CSP modelchecker FDR (which we have done for our example). The second CSP refactoring we use in the example is "Pull Up CSP". This refactoring is described by the following template (with empty condition).

BEFORE:

$super_{before}$
`main` $\stackrel{c}{=} R$

$sub_{before}+$
`inherit` $super_{before}$
`main` $\stackrel{c}{=} P_{sub\ Chans(P)} \|_{Chans(Q)} Q$

AFTER:

$super_{after}$

$\texttt{main} \stackrel{c}{=} Q_{Chans(Q)} \|_{Chans(R)} R$

$sub_{after}+$

$\texttt{inherit}\ super_{after}$

$\texttt{main} \stackrel{c}{=} P_{sub}$

The template assumes to have a superclass $super_{before}$ and a (nonzero) number of subclasses sub_{before} which all have their CSP processes defined as the parallel composition of some specific process P_{sub} and one joint process Q. The process Q can then be pulled upwards to the superclass.

We can prove the correctness of this refactoring again by looking at the CSP part in isolation. We thus simply prove that the CSP part before and after applying the refactoring is the same. The proof uses the definition of the semantics of inheritance (parallel composition of sub- and superclass).

$$
\begin{aligned}
& procC(sub_{before}) = (P_{Chans(P)} \|_{Chans(Q)} Q)_{Chans(P \| Q)} \|_{Chans(R)} R \\
= \quad & \{Chans(X_{Chans(X)} \|_{Chans(Y)} Y) = Chans(X) \cup Chans(Y)\} \\
& (P_{Chans(P)} \|_{Chans(Q)} Q)_{Chans(P) \cup Chans(Q)} \|_{Chans(R)} R \\
= \quad & \{ \|_x \|_y - \text{assoc from [Ros97]}\} \\
& P_{Chans(P)} \|_{Chans(Q) \cup Chans(R)} (Q_{Chans(Q)} \|_{Chans(R)} R) \\
= \quad & \{Chans(X_{Chans(X)} \|_{Chans(Y)} Y) = Chans(X) \cup Chans(Y)\} \\
& P_{Chans(P)} \|_{Chans(Q \| R)} (Q_{Chans(Q)} \|_{Chans(R)} R) \\
= \ & procC(sub_{after})
\end{aligned}
$$

Extract Superclass. Finally, we show correctness of a refactoring which changes both CSP and Object-Z part of a class. "Extract Superclass" is a complex refactoring. First we introduce an empty superclass, then we use the refactorings "Pull Up Method", "Pull Up Field" and "Pull Up CSP". The latter three are also normal refactorings which we have already treated above. Therefore, we only have to prove correctness of the introduction of a new empty superclass (with template given below).

BEFORE:

sub_{before}

$\texttt{main} \stackrel{c}{=} PE$

AFTER:

$super_{after}$

$\texttt{main} \stackrel{c}{=} Skip$

sub_{after}

$\texttt{inherit}\ super_{after}$

$\texttt{main} \stackrel{c}{=} PE$

Fortunately, this new superclass does not add new functionality, so we only have to prove that both the CSP and Object-Z part will not be changed. First we prove this for the CSP part:

$$\begin{aligned} procC(sub_{after}) &= PE \,{}_{Chans(PE)}\|_{Chans(Skip)}\, Skip \\ &= PE \,{}_{Chans(PE)}\|_{\{\checkmark\}}\, Skip \\ &= PE \equiv procC(sub_{before}) \end{aligned}$$

For the Object-Z part correctness trivially holds since the superclass does not introduce new constraints on the Object-Z part. Thus we have proven that the introduction of an empty superclass is behaviour preserving as well, which finishes the correctness proofs for the refactorings of our example. These proofs exemplarily show all possible types of correctness proofs for CSP-OZ refactorings: most of them proceed by syntactically rewriting of (state, init or operation) schemas or CSP processes. Some of them, however, explicitly need the construction of a refinement relation.

5 Conclusion

In this paper we have shown how to carry out refactorings in object-oriented specifications involving a state-based as well as a behaviour-oriented part. Refactorings thus concerned either only one of the specification parts (CSP or Object-Z) or both. We have shown correctness of (some of) these refactorings by proving a refinement relationship between before and after specification. This guarantees the desired behaviour preservation.

Related work. Refactoring is a widely used technique in program design and development. An overview over different approaches to refactoring is given in [MT04]. The use of data refinement as a correctness criterion for refactorings is also followed in Cornèlio, Cavalcanti et. al. [Cor04,CCS02] and McComb [MS04,McC04]. Cornèlio defines refactorings and proves their correctness for a refinement-based object-oriented language (ROOL), McComb and Smith use Object-Z. While in particular the latter approach is close to ours, both languages are state based formalisms only and do not include dynamic aspects, like CSP-OZ does. A different approach to correctness of refactorings is taken by Bannwart and Müller [BM06b]. They show that particular pre- and post-conditions can be derived from a refactoring and used to ensure correctness by inserting them as assertions into programs. Then they are able to implement a runtime check of the correctness of refactorings.

A frequently used formal approach to refactorings is the application of graph transformations (e.g. [HT04,KHE03,MEDJ05,SD06,BM06a]). Graph transformation rules can be used to describe refactorings when the specification can be seen as a graph (e.g. in case of UML diagrams). They however cannot deal with data-specific conditions, and most often do not treat different views, like the data and process view we have here.

References

[BM06a] T. Baar and S. Markovìc. A Graphical Approach to Prove the Semantic Preservation of UML/OCL Refactoring Rules. Technical report, Ecole Polytechnique Fédérale de Lausanne, 2006.

[BM06b] F. Bannwart and P. Müller. Changing programs correctly: Refactoring with specifications. In *FM*, number 4085 in LNCS, pages 492–507. Springer, 2006.

[CCS02] M. L. Cornélio, A. L. C. Cavalcanti, and A. C. A. Sampaio. Refactoring by Transformation. In *REFINE'2002*, volume 70 of *Eletronic Notes in Theoretical Computer Science*. Elsevier, 2002.

[Cor04] Márcio L. Cornèlio. *Refactorings as Formal Refinment*. PhD thesis, Universidade Federal de Pernambuco, 2004.

[DB01] J. Derrick and E. A. Boiten. *Refinement in Z and Object-Z*. Springer, 2001.

[dE98] W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. CUP, 1998.

[DW06] J. Derrick and H. Wehrheim. Model transformations incorporating multiple views. In *AMAST*, volume 4019 of *LNCS*, pages 111–126. Springer, 2006.

[FDR97] Formal Systems (Europe) Ltd. *Failures-Divergence Refinement: FDR2 User Manual*, Oct 1997.

[Fis97] C. Fischer. CSP-OZ: A combination of Object-Z and CSP. In *FMOODS '97*, volume 2, pages 423–438. Chapman & Hall, 1997.

[Fis00] C. Fischer. *Combination and Implementation of Processes and Data: from CSP-OZ to Java*. PhD thesis, University of Oldenburg, 2000.

[Fow04] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 2004.

[Hoa85] C.A.R. Hoare. *Communicating sequential processes*. Prentice Hall, 1985.

[HT04] R. Heckel and S. Thöne. Behavior-preserving refinement relations between dynamic software architectures. In *17th Int. Workshop on Algebraic Development Techniques*, pages 1–27, 2004.

[KHE03] J. Küster, R. Heckel, and G. Engels. Defining and validating transformations of UML models. In *HCC*, pages 145–152. IEEE Computer Society, 2003.

[McC04] T. McComb. Refactoring Object-Z Specifications. In *FASE'04*, LNCS, pages 69 – 83. Springer, 2004.

[MEDJ05] T. Mens, N. Van Eetvelde, S. Demeyer, and D. Janssens. Formalizing refactorings with graph transformations. *Journal of Software Maintenance*, 17(4):247–276, 2005.

[MS04] T. McComb and G. Smith. Architectural Design in Object-Z. In *Australian Software Engineering Conference (ASWEC'04)*, pages 77 – 86. IEEE Computer Society Press, 2004.

[MS06] T. McComb and G. Smith. Refactoring object-oriented specifications: A process for deriving designs. Technical Report SSE-2006-01, University of Queensland, Australia, May 2006.

[MT04] T. Mens and T. Tourwé. A survey of software refactoring. *IEEE Trans. Software Eng.*, 30(2):126–139, 2004.

[Ros97] W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997.

[Ruh06] Th. Ruhroth. Refactoring Object-Z Specifications. In *18th Nordic Workshop on Programming Theory*, 2006.

[SD06] R. Van Der Straeten and M. D'Hondt. Model refactorings through rule-based inconsistency resolution. In J. Bézivin, editor, *Proceedings of the 2006 ACM Symposium on Applied Computing*, pages 71210–1217, 2006.

[Smi00] G. Smith. *The Object-Z Specification Language*. KAP, 2000.