Model transformations across views

John Derrick\textsuperscript{a}, Heike Wehrheim\textsuperscript{b,*}

\textsuperscript{a} Department of Computing, University of Sheffield, Sheffield, S1 4DP, UK
\textsuperscript{b} Universität Paderborn, Institut für Informatik, 33098 Paderborn, Germany

\textbf{A B S T R A C T}

Models of software often describe systems by a number of (partially) orthogonal views: a state machine, a class diagram, a scenario might specify different aspects of the one system to be built. Such abstract, multi-view models are the starting point for transformations into platform-specific models and finally the code. However, during these transformations it is usually not possible to keep such a neat separation into different views: the specification language of the target models might not support all such views. The target model, however, still needs to preserve the behaviour of the abstract, multi-view model. Therefore, model transformations have to be capable of moving aspects of the behaviour across views.

In this paper, we study model transformations migrating aspects from state-based views (i.e., class specifications with data and methods) to protocol-based views (i.e., process specifications on orderings of methods) and vice versa. The specification languages for these two views are equipped with a \textit{joint, formal} semantics which enables a proof of behaviour preservation. We consequently derive conditions for our transformations to be behaviour-preserving, where behaviour preservation is characterised by \textit{refinement}.

\section{Introduction}

Model-based design of software systems advocates the construction of \textit{models} preceding an actual implementation. In this approach, \textit{model transformations} are used to bridge the gap between high-level, platform-independent and low-level, platform-specific models close to an implementation. The most frequently used modelling language, which has also gained industrial acceptance, is currently the Unified Modelling Language (UML). The UML comprises a number of different diagrammatic notations for describing certain views of systems, for instance class diagrams for a static view of classes and their relationships, sequence diagrams for scenarios of a system, state machines for protocols of method executions (the dynamic behaviour) or component diagrams for an architectural view. Implementation languages on the other hand usually do not reflect all of these views: they might not have a separation of static view and dynamic behaviour, no means of describing scenarios, or might even not be based on the object-oriented paradigm that UML favours. Model transformations thus need to move behaviour descriptions from one view to another, or merge two views into one, while still preserving the abstractly specified behaviour.

In this paper, we will be concerned with such \textit{view-crossing} model transformations. More specifically, we will look at models consisting of two views: a state-based view describing classes with their attributes and methods, and a process-oriented view modelling orderings of method executions. The model transformations move (parts of the) behaviour...
descriptions between these views, thus making it possible to describe aspects in different ways. In particular, model transformations can merge two views into one as well as split a single view into two. However, we can — of course — not allow for arbitrary transformations. The general condition on transformations is always behaviour preservation: the target model of a transformation is required to preserve the behaviour of the source model. For a user of the system, an interaction with the target model (a use of a system implemented according to the target model) should not be distinguishable from an interaction with the source model. Behaviour preservation should guarantee substitutability; the final implementation must not deviate from the abstractly modelled behaviour.

Here, we develop model transformations which provably guarantee this type of behaviour preservation. To this end, we use modelling languages for our views which are equipped with a formal semantics. A semantics of the individual formalisms alone is however not sufficient, we furthermore need a semantics for their combined use as the views together form the system specification. Moreover, there must be a clean concept of behaviour preservation (or substitutability) defined on the joint semantic domain. A modelling language which fulfills all these requirements is the integrated formal notation CSP–OZ, a combination of the process algebra CSP and the object-oriented specification formalism Object-Z. While Object-Z allows for the modelling of state-based aspects (like class diagrams), CSP can be used to specify method orderings (similar to state machines). Both formalisms have a semantics of their own and along with the semantics a concept of refinement (data and process refinement, respectively). Both notions of refinement guarantee the desired substitutability. Moreover, refinement is compositional: individual refinements on the static Object-Z and the dynamic CSP parts lead to a refinement on their combination.

Taking CSP–OZ and the refinement concepts of CSP and Object-Z as our starting point, we investigate transformations crossing the border between the Object-Z and CSP parts. More specifically, we look at transformations which move parts of the behaviour specified in Object-Z into the CSP part, and vice versa. Due to the requirement of behaviour preservation, such transformations are valid only if they constitute a refinement in the combined model. This leads us to develop a number of conditions on transformations which guarantee behaviour preservation. A formal proof of soundness of these conditions based on the existing semantics is given.

The transformations are flexible enough that they could be used in a development process that requires either the addition or removal of particular constraints as one moves through the development cycle, and indeed the steps could be combined if necessary. The general approach is exemplified using two larger case studies, and a simple example of a buffer specification illustrates the more technical aspects.

The structure of the paper is as follows. In Section 2, we introduce one of the case studies in order to motivate the discussion, and this is followed by a description of the semantics of the CSP–OZ notation as well as how refinement is defined in it (Section 3). In Section 4, we derive the conditions on transformations which guarantee behaviour preservation, and these are illustrated on the case studies in Section 5. We conclude in Section 6.

2. Motivation

Our first case study is used to introduce the modelling formalism considered here and to discuss view-crossing transformations. In general, we envisage our approach to be applied in a model-driven development (MDD) which incrementally constructs several models with various levels of detail. Starting with a first platform-independent model, this is transformed to a platform-specific model and finally into an implementation. The aspect of interest for us in MDD is the question of correctness of the applied model transformations, i.e., whether lower level models adhere to the specifications given in earlier design steps. Our case study thus consists of two models, a platform-independent and a platform-specific one, and alongside these two we discuss the question of correctness.

The case study concerns the modelling of a manufacturing system, a system that processes workpieces (initially stored in some in-store) by a number of machines and finally stores them in an out-store. The transportation of workpieces in the factory is carried out by autonomous (also called holonic) transportation units (abbreviated HTS = holonic transportation systems). Transportation is not regulated by a central control but achieved via negotiation between stores, machines and HTS. Once a store or machine has a workpiece to be transported, it asks the transportation units for offers about the costs of such a transportation. Having obtained all offers, it chooses the one with the smallest cost and orders it. We illustrate our ideas using one part of the case study concerning this negotiation. We further simplify the specification to ease readability by assuming that the manufacturing system knows how many transportation systems there are, and waits for all of them to return an offer before choosing which one to accept.

2.1. Model 1 — Platform-independent

The initial abstract model consists of two views: A static view describing (just one) class with attributes and methods, and a dynamic view describing a protocol of interactions for this class. With the terminology “static” and “dynamic” we follow the often used classification of UML diagrams into static and dynamic ones. Thus our state-based part of the specification is in the following often referred to as static view.
2.1.1. Static view

The static view is given in Object-Z [35]. The role of Object-Z in our model is similar to that played by class diagrams with OCL constraints in UML models: it describes variables and methods of classes with class invariants and pre- and postconditions. A class specification starts with a definition of the interface of the class, followed by a declaration of its state (its variables) together with an initialisation schema giving restrictions on the initial values of variables.

After that, a number of operation schemas define the methods, or operations, of the class. Here, primed and unprimed variables refer to the after and before states, respectively; input and output parameters are decorated with ? and ! respectively, and are undecorated if they are used for addressing objects. The predicates appearing in the schemas of methods state pre- and postconditions of methods (in UML, this would be specified in OCL). The precondition acts as a guard for method execution; outside the precondition method execution is refused (i.e., it is blocked).

We start with the specification of the set of transportation agents involved in the system. Note that this information would normally be derived from a component diagram specifying the architecture of the system. To avoid introducing a third view in our specification, we give it here directly.

\[ Hts \equiv \{h_1, h_2, h_3\} \]

Next, we give an Object-Z class specification for the machines.

```
 Machine
 method offer : [ h : Hts; cost? : Cost ]
 method order : [ h : Hts ]
 local_chan choose
 ...
 offers : seq(Hts × Cost)
 orderTo : Hts
 ...

 offer
 \[ \Delta(offers) \]
 \[ h : Hts; cost? : Cost \]
 \[ offers' = offers \wedge ((h, cost?)) \]

 order
 \[ \Delta(offers) \]
 \[ h : Hts \]
 \[ h = orderTo \wedge offers' = \{\} \]

 choose
 \[ \Delta(orderTo) \]
 \[ \exists i : 1..#offers \cdot n : Cost \bullet \]
 \[ offers i = (orderTo', n) \wedge \]
 \[ \forall j : 1..#offers \bullet n \leq \text{second}(offers j) \]
 ...
```

The class Machine includes two public methods offer and order and a private method choose. It has a variable offers storing the name of each offering HTS together with the cost, given here as a sequence of pairs. This reflects the option we have chosen here for allowing arbitrary many offers. It also has a variable orderTo describing the chosen HTS. The type Cost is a fixed range of natural numbers. Method offer changes the variable offers (denoted by having offers in its delta-list), and appends the next offer to the end of the sequence. Method order orders the transportation unit, which is currently assigned to attribute orderTo and empties the sequence offers. Method choose chooses an HTS with the cheapest offer in the sequence and assigns it to orderTo. The static view also contains a number of classes for the stores and the transportation system, these are elided here. A more complete specification can be found in [39].

2.1.2. Dynamic view

In the dynamic view every class in the system may, in addition, have a protocol regulating the allowed ordering of method invocation. For class Machine, it is given as the following CSP process description.

```
 main = FindHts; main
 FindHts = (||| h:Hts offer.h \rightarrow \text{SKIP}); choose \rightarrow order \rightarrow \text{SKIP}.
```

This process description specifies the protocol for class Machine to be a repeated execution of getting offers, followed by choosing an offer and ordering it. In the CSP description, main defines the dynamic behaviour of Machine, ||| is the parallel interleaving operator, offers are obtained from the HTS in parallel; \(\rightarrow\) and \(;\) are sequencing operators, and \text{SKIP}
denotes termination. Again, we elide methods of Machine that were not included in the above class specification (e.g., those concerned with loading, unloading and processing workpieces).

2.1.3. Integration

The complete model consists of the static and dynamic views. Intuitively, the semantics of this combination can be understood as conjunction: both the Object-Z and the CSP part impose certain restrictions on the behaviour of the system (viz. the execution of its methods and the associated state change). In the integration, both these restrictions have to be obeyed. A formal definition of the semantics is given in Section 3.

Note that in CSP–OZ specifications the CSP part cannot refer to the variables of the Object-Z part (the semantics of the integration is defined by parallel composition) and the CSP part is mainly responsible for regulating the data-independent aspects of the specification. Thus, whenever we have an operation name only in a CSP process (or, more generally, an operation name with an incomplete number of parameters with respect to those specified in the interface), the CSP part will allow any value for parameters. Thus, e.g., order in the above process description is an abbreviation for order?x.

2.2. Model 2 – Platform-specific

Next, we want to modify our first model in order to improve its implementability in a programming language, i.e., we make a transformation towards a platform-specific model. The target language in the project [26] that motivated this work was Java, however, here we just assume that our target language requires

- methods to be deterministic, and
- data types to be bounded.

This necessitates two changes in our model. The private method choose is nondeterministic, since it chooses an arbitrary HTS out of those with the smallest cost. Furthermore, the sequence offers used to store offers coming from transportation units is unbounded. We thus need the following transformations:

(1) Determinism. The change necessary to make choose deterministic is the addition of \( n = \text{second}(\text{offers}\, j) \Rightarrow i < j \) to its predicate. If more than one HTS has given a minimal offer, the first one in the sequence offers is chosen.

(2) Bounded data types. In addition, we have to fix an upper bound for the size of offers. We do so by adding a class invariant to the specification restricting the size of offers to three (we will soon see why three is sufficient): \#offers \( \leq 3 \). Note that we thus implicitly specify method offer to be blocked once the sequence has reached size three.

The dynamic view remains unchanged. The resulting platform-specific model thus consists of the following new class specification together with the same CSP process main.

```plaintext
Machine2
...
offers : seq(Hts \times Cost)
orderTo : Hts
...
#offers \leq 3
...
choose

\[ \Delta(\text{orderTo}) \]
\[ \exists i : 1..#offers; n : \text{Cost} \quad \bullet \quad
\text{offers}\, i = (\text{orderTo}', n) \quad \wedge
\forall j : 1..#offers \bullet n \leq \text{second}(\text{offers}\, j)
\quad \wedge (i \neq j \wedge n = \text{second}(\text{offers}\, j)) \Rightarrow i < j \]
...
```

2.3. Behaviour preservation?

This completes the specification of our platform-independent and platform-specific models. Next, we are interested in showing the correctness of the employed model transformation. However, when just looking at the Object-Z part, we see that the transformation is not behaviour-preserving. The addition of the class invariant \#offers \( \leq 3 \) invalidates substitutability,
whereas offer was executable in all states in the first model, it now might be blocked (when the sequence of offers has reached the length 3). A user of the two models can actually detect a difference.

Fortunately, this transformation is still sound overall as this additional restriction in the Object-Z part was already present in the CSP part of the first model. Remember that the CSP part restricted the ordering of method executions to the following: first getting 3 offers (from the 3 HTS), then choosing an offer, ordering it and starting all over again. Thus, the sequence offers will never be empty when choose is executed, and always empty when offer is executed. The additional constraints in the Object-Z part just made some restrictions explicit in the state-based view, which were already present in the dynamic view. Therefore, in this transformation we have added parts of the behaviour specification of the dynamic view to the static view — it is thus a view-crossing transformation. In the following, we need to precisely define under what conditions such transformations are sound.

3. Background

In order to study the soundness of transformations, we first need to define the semantics of our specification formalism, and state what behaviour preservation actually means. The integrated formal method CSP–OZ [14] is a combination of Object-Z and CSP and inherits most of their theory. Central to both formalisms is a notion of refinement. In Object-Z, data refinement is concerned with changing data structures and operations while preserving the externally visible behaviour; and in CSP, failures refinement deals with reducing nondeterminism in abstract process specifications. As is normal in formal approaches to model transformation, these two concepts will be used as our criteria for behaviour preservation.

3.1. Semantics

In this paper, we are interested in a joint semantics for Object-Z and CSP in which we carry out proofs of behaviour preservation. To this end, we use an operational semantics given in terms of labelled transition systems. Transition systems describe states of a system and transitions between states which are labelled with the events taking place during the state change. Here, events consist of the names of operations $O_1, O_2, \ldots \in OP$ plus possible values of input and output parameters $i_1, i_2, \ldots \in In, o_1, o_2, \ldots \in Out$. Thus, $O_1.i_1.o_1$ is a typical event; the set of such visible events is Event. In addition, $\tau$ denotes the internal, invisible event, and we let $Event_{\tau} = Event \cup \{\tau\}$. For a set of operation names $O \subseteq OP$, following FDR’s notation we let $\{O\}$ denote the set of events over these operations.

The following defines labelled transition systems, which will be the semantic domain for both CSP and Object-Z as well as their integration.

Definition 1. A labelled transition system $T = (Q, Q_0, E, \rightarrow)$ consists of

- a set of states $Q$,
- a set of initial states $Q_0 \subseteq Q$,
- an alphabet $E \subseteq Event_{\tau}$, and
- a transition relation $\rightarrow \subseteq Q \times E \times Q$. □

3.1.1. CSP

For CSP, the transition system is derived from the standard structural operational semantics [30], and thus here we only give an example. We restrict the use of CSP by excluding hiding and internal choice so as to get a divergence-free process.

As an example, consider the CSP buffer process specification (part of the example in the next section):

\[
\text{main} = \text{put} \rightarrow \text{get} \rightarrow \text{main}.
\]

Again, this is shorthand for main = put?x → get?x → main. Assuming that put has one input and get one output (which is fixed in the interface of a class), and that the base set of elements is $\{1, 2\}$, this process will define the following transition system:

Here, the states of the system are the evolutions of the behaviour expression, so, for example, the initial state is main and after an event put.1 the system evolves to the state get → main. It can be seen that the CSP part just regulates the ordering of puts and gets but does restrict the data values.
3.1.2. Object-Z

Recall that an Object-Z class consists of a state schema State, an initialisation schema Init and a number of operation schemas Opj. Intuitively, the state schema determines the set of possible states (namely, all possible valuations of variables vars(State) being declared in State), the initialisation schema determines the initial states and the operation schemas the transitions. However, we have to take some care in our treatment of inputs and outputs, and thus the transition system semantics of the Object-Z part is slightly more complicated than that of CSP.

Let us start with the states. The state schema defines the variables with their types. Assuming that variables take values from some global domain D covering all types, a valuation or binding of variables is a mapping \( z : \text{vars}(\text{State}) \rightarrow D \). Variables in state schemas are declared with a specific type and we assume bindings to be type-correct, i.e., only assign values of corresponding types to the variables. We let \( \text{Bind}(\text{State}) = \{ z : \text{vars}(\text{State}) \rightarrow D \mid \text{State}\} \) be the set of all type-correct bindings of a state schema State. To facilitate reading, we often omit State and just write \( \text{Bind} \). The set of initial states is thus \( \{ z : \text{Bind} \mid \text{Init}\} \). An operation \( \text{Op} \) is enabled in some binding and for some particular input value if the predicate in the operation schema \( \text{Op} \) can be made true for some next state binding and output. If there is more than one such next state binding and output, these are chosen nondeterministically. This latter fact makes the operational semantics a little more complicated.

Nondeterministic choices need to be modelled by \( \tau \)-transitions in the operational semantics so as to match the failures-divergences semantics of CSP–OZ given in [15]. Thus, from a particular binding there is first a \( \tau \)-transition to a state in which the choice of next state and output has been taken for all enabled operations. Only in this state, visible operations are executed. The intermediate states are modelled by mappings

\[
(\text{Bind} \times \text{OP} \times \text{In}) \rightarrow (\text{Bind} \times \text{Out})
\]

For the current binding, one specific operation and input, the mapping determines the next state binding and output. The mapping has to be consistent with the specification of the operation, i.e., if \( (z, \text{Op}, i, z', o) \in Z \) for some \( Z : (\text{Bind} \times \text{OP} \times \text{In}) \rightarrow (\text{Bind} \times \text{Out}) \), then the predicate in the schema of \( \text{Op} \) is satisfied for the current and next state, input and output: \( (z, i, o, z') \in \text{Op} \). Since there are operations which might not be enabled in the current state, this mapping is given as a partial function. The following then defines the labelled transition system of an Object-Z class.

**Definition 2.** Let \( C = (\text{State}, \text{Init}, (\text{Op}_j)_{j\in J}) \) be an Object-Z class. Its labelled transition system \( (Q, Q_0, E, \rightarrow) \) is defined as

- \( Q = \text{Bind} \cup \{ z : (\text{Bind} \times \text{OP} \times \text{In}) \rightarrow (\text{Bind} \times \text{Out}) \mid \forall (z, \text{Op}, i, z', o) : Z \bullet (z, i, o, z') \in \text{Op}, \}
- \( Q_0 = \{ z : \text{Bind} \mid \text{Init}\}, \)
- \( E = \{ \mid \text{Op}_j \mid j \in J \mid \}, \)
- for \( z : \text{Bind}, Z : (\text{Bind} \times \text{OP} \times \text{In}) \rightarrow (\text{Bind} \times \text{Out}), \text{Op} : \text{OP}, i : \text{In} \) and \( o : \text{Out} \) we have
  - \( z \xrightarrow{\tau} Z \) iff \( \forall (z_1, \text{Op}, i, z_2, o) : Z \bullet z = z_1 \)
  - \( Z \xrightarrow{\text{Op}_j.o} z \) iff \( \forall (z_1, \text{Op}', i', z_2, o') : Z \bullet (\text{Op} = \text{Op}' \land i = i') \Rightarrow (o = o' \land z = z_2). \)

As an example, consider the following buffer specification, where \( \text{Elem} = \{1, 2\} \) is the base set of elements.

\[
\begin{array}{|c|c|}
\hline
\text{OZBuf} & \text{Init} \\
\hline
\text{buf} : \text{seq} \text{Elem} & \text{buf} = () \\
\hline
\text{put} & \Delta(\text{buf}) \\
\text{i?} : \text{Elem} & \text{get} \\
\text{buf}' = (\text{i?}) \sim \text{buf} & \Delta(\text{buf}) \\
\hline
\#\text{buf} > 0 & \text{o!} : \text{Elem} \\
\text{buf} = \text{buf}' \sim \langle\text{o!}\rangle \\
\hline
\end{array}
\]

In contrast to the buffer described by the CSP example above, its transition system is infinite since the buffer can store an arbitrary number of elements. Here, in its transition system, we see the correct processing of data: after putting in element 1, only event get.1 is possible, not get.2.
An example binding is given by the initial state which is \( \{\text{buf} \mapsto \langle \rangle\} \). An example state is the one after the first \( \tau \) transition, where the system reaches the state \( \{(\text{buf} \mapsto \langle \rangle), \text{put}_1, \{\text{buf} \mapsto \langle 1\rangle\}, \bot\}, \{(\text{buf} \mapsto \langle \rangle), \text{put}_2, \{\text{buf} \mapsto \langle 2\rangle\}, \bot\} \), where \( \bot \) stands for no output.

3.1.3. Integration

The semantics of the combination of the CSP and Object-Z parts is simply the parallel composition of the transition systems, synchronising on joint events. The following is a general definition of parallel composition, with a free choice of the synchronisation set. In the case of CSP–OZ, the synchronisation set \( S \subseteq \text{Event} \) a synchronisation set.

**Definition 3.** Let \( T_i = (Q_i, Q_{0,i}, E_i, \rightarrow_i), i = 1, 2 \), be two labelled transition systems and \( S \subseteq \text{Event} \) a synchronisation set. The parallel composition \( T_1 \parallel S T_2 \) is defined as \( T = (Q, Q_0, E, \rightarrow) \) with

- \( Q = Q_1 \times Q_2 \),
- \( Q_0 = Q_{0,1} \times Q_{0,2} \),
- \( E = E_1 \cup E_2 \), and
- \( (q_1, q_2) \rightarrow (q'_1, q'_2) \) iff
  - \( \exists ev \in S \land q_1 \rightarrow_{ev} q'_1 \land q_2 \rightarrow_{ev} q'_2 \), or
  - \( \exists ev \notin S \land (q_1 \rightarrow_{ev} q'_1 \land q_2 = q'_2) \lor (q_2 \rightarrow_{ev} q'_2 \land q_1 = q'_1) \).

The combination of the above CSP and Object-Z specification, defined as a parallel composition with synchronisation set \( \{\text{put}_1, \text{put}_2, \text{get}_1, \text{get}_2\} \), thus gives us the following transition system:

In the combination, the specification has the behaviour of a proper one-place buffer: \( \text{put} \) and \( \text{get} \) occur in turns, and only elements which have been put in can be removed. The two views place different restrictions on the system which are conjoined in the semantics.

3.2. Refinement

The semantic domain of transition system is also the basis for our definition of behaviour preservation. Behaviour preservation should guarantee substitutability, i.e., for an external observer the lower level model should be indistinguishable from the higher level model. In formal methods, this is captured by the notion of refinement [9]. For CSP, we use refinement based upon a process’ stable failures [30], for Object-Z we use data refinement [6]. We begin with an explanation of stable failures refinement.

3.2.1. Stable failures refinement

There are a number of different refinement preorders defined on CSP processes. Since our systems do not diverge, a natural choice is to use a refinement relation based upon a process’ stable failures. The alternative would be to use the failures–divergences refinement relation, however, due to the blocking semantics of Object-Z, the static view does not give rise to any potential divergence, and neither does the CSP part, thus it is sensible to concern ourselves with the stable failures of a system. We use the following notation and definition.

**Definition 4.** Let \( T = (Q, Q_0, E, \rightarrow) \) be a transition system labelled over the alphabet \( \text{Event}_\tau \).

1. For states \( q, q': Q, ev : \text{Events}_\tau \), the notation \( q \rightarrow ev \rightarrow q' \) stands for \( q \rightarrow^*_\tau ev \rightarrow^*_\tau q' \) (where \( \rightarrow^*_\tau \) is the reflexive and transitive closure of \( \rightarrow^*_\tau \)).
2. For a state \( q : Q, ev : \text{Event}_\tau \), \( q \rightarrow ev \) stands for \( \exists q' : Q \bullet q \rightarrow ev q' \).
(3) For a state $q : Q$, $ev : Event$, $q \xrightarrow{ev}$ stands for $\neg \exists q' : Q \bullet q \xrightarrow{ev} q'$.

(4) For states $q, q' : Q$, $ev_i : Event$, $i = 1, \ldots, n$, we define $q \xrightarrow{ev_1, \ldots, ev_n} q'$ iff there are states $q_0, q_1, \ldots, q_n : Q$ such that $q = q_0, q_i \xrightarrow{ev_{i+1}} q_{i+1}, 0 \leq i \leq n - 1$, and $q_n = q'$.

(5) For a state $q : Q$, a sequence of events $tr : Event^*$, the notation $q \xrightarrow{tr}$ stands for $\exists q' : Q \bullet q \xrightarrow{tr} q'$.

(6) A state $q : Q$ is stable iff $q \xrightarrow{\omega}$.

(7) The set of initial events $init(q)$ of a state $q : Q$ is

$$init(q) \triangleq \{ ev : Event \mid q \xrightarrow{ev} \}.$$ 

(8) The set of traces of $T$ is

$$traces(T) \triangleq \{ tr : Event^* \mid q_0 \xrightarrow{tr} \wedge q_0 \in Q_0 \}.$$ 

(9) The set of stable failures of $T$ is

$$failures(T) \triangleq \{ (tr, X) : Event^* \times I Event \mid \exists q : Q : \exists q_0 : Q_0 \bullet q_0 \xrightarrow{tr} q \wedge (q \text{ stable}) \wedge (X \cap init(q) = \emptyset) \}.$$ 

(10) $\mathcal{F}(T) \triangleq (traces(T), failures(T))$. $\square$

We adopt the convention that if the transition relation $\rightarrow$ is indexed with the name or numbering of the transition system it belongs to, e.g., $\rightarrow_1$ or $\rightarrow_A$, then the derived relation $\Rightarrow$ will also have the same index, i.e., $\Rightarrow_1$ or $\Rightarrow_A$.

Intuitively, the stable failures contain pairs $(tr, X)$ such that the transition system can refuse to perform the events in $X$ after having executed the trace $tr$. Failures are only computed in stable states in which no internal behaviour of the system is possible anymore. For example, the stable failures of the CSP process main above include: $(ev, \{get.1, get.2\})$, $(put.1, \{put.1, put.2\})$, $(put.2, \{put.1, put.2\})$, …

Refinement in this stable failures model compares the traces and failures of two models: after having executed the same trace, a lower level model may not refuse events which the higher level model allowed. If it does so, then this would be an externally visible difference and refinement would not hold between the two levels.

**Definition 5.** Let $T_1, T_2$ be labelled transition systems.

$T_2$ is a (stable) failures refinement of $T_1$, $T_1 \sqsubseteq_A T_2$, iff $\mathcal{F}(T_2) \subseteq \mathcal{F}(T_1)$.

$T_2$ and $T_1$ are failures equivalent, $T_2 \equiv_A T_1$, iff $\mathcal{F}(T_2) = \mathcal{F}(T_1)$. $\square$

Having introduced these definitions at the level of transition systems, these notions are thus applicable to both the CSP and Object-Z parts. For Object-Z, there is a further way of comparing two specifications without looking at their operational semantics, which we now explain.

### 3.2.2. Data refinement

Data refinement in Object-Z is concerned with showing that a change of data structures and operations still guarantees behaviour preservation [6]. To this end, the variables of abstract and lower level model are related via a retrieve (or representation) relation $R$. The representation relation describes the correspondence between abstract and concrete states. Given such a correspondence, refinement can be shown using the following downward simulation conditions.

**Definition 6.** Let $A = (AState, AInit, (AOp_j)_{j \in J})$ and $C = (CState, CInit, (COp_j)_{j \in J})$ be Object-Z classes. $C$ is a downward simulation of $A$, $A \sqsubseteq_d C$, iff the following hold for all $j \in J$:

- $\forall AState \bullet CInit \Rightarrow \exists AState \bullet AInit \wedge R,$
- $\forall AState ; AState \bullet R \Rightarrow (\pre AOp_j \Leftrightarrow \pre COp_j),$
- $\forall AState ; CState'; AState \bullet R \wedge COp_j \Rightarrow \exists AState' \bullet R' \wedge AOp_j$. $\square$

These conditions essentially say that the steps of the lower level systems $C$ are a simulation of the system $A$, and whenever an operation is enabled in $A$ it should also be enabled in $C$ (and vice versa). Here, $\pre AOp$ refers to the precondition of an operation, i.e. the guard of an operation. Formally, the precondition is defined as

$$\pre AOp \triangleq \exists AState' ; \text{sf} : \text{Out} \bullet AOp.$$ 

For instance, the precondition of operation get is $\pre \text{get} = (\text{buf} \neq \{\})$. Preconditions may neither be weakened nor strengthened in an Object-Z data refinement.
3.2.3. Integration

Work relating data refinement with refinement in CSP includes Josephs [20], He [19], Bolton and Davies [2,3], Derrick and Boiten [1,7] and Schneider [34]. In [20] and [19], a basic correspondence between downward (and upward) simulation rules and failures-divergences refinement is defined. The more recent work of Bolton and Davies [2,3], Derrick and Boiten [1,7] and Schneider [34] includes specific consideration of input and output which introduces some subtleties. A survey of the relevant results is given in [8].

The result applicable in our context is the following [8], which shows that the downward simulation between Object-Z specifications implies failures refinement on their transition systems.

Lemma 1. Let $A$, $C$ be Object-Z specifications with transition systems $T_A$ and $T_C$. Then:

$$A A C \Rightarrow T_A \subseteq_f T_C.$$

Furthermore, stable failures refinement is preserved under parallel composition [30]: all CSP operators are monotonic under the stable failures order. This applies to alphabetised as well as general parallel composition.

Lemma 2. Let $T$, $T_1$, $T_2$ be transition systems, $S \subseteq$ Event an arbitrary set of events. Then:

$$T_1 \subseteq_f T_2 \Rightarrow T_1 ||s T \subseteq_f T_2 ||s T \land T ||s T_1 \supseteq_f T ||s T_2.$$

Lemmas 1 and 2 together show that an individual refinement of one of the views also gives us a refinement of the combined system. This already provides us with a technique for proving behaviour preservation of views in isolation. However, there are valid transformations which cannot be verified in isolation in the individual views. The example introduced above already illustrates this. There, we were interested in two transformations:

1. Determinism. The first change made a method deterministic. Since nondeterminism was reduced in the transformation, this change can be verified by a refinement of an individual view.
2. Bounded data types. By restricting the length of the sequence offers we have made an observable change here, and this could not be verified as a valid refinement, since this alters the preconditions of the methods.

Thus, of these changes, only the first is covered by transformations which are refinements of the individual viewpoints. In the next section, we will begin studying techniques to overcome cases where such separate proofs of behaviour preservation are not possible.

4. Transformations across views

We first consider a simpler example, on which most of our transformations can be exemplified. Once the basic concept is clear, we will develop conditions for showing that such transformations are sound.

4.1. Example

As our running example, we take the following specification of a one-place-buffer, partly seen already in the last section.\(^1\)

[Elem]

Buffer

\[
\begin{array}{l}
\text{method } \text{put} : [i? : \text{Elem}] \\
\text{method } \text{get} : [o! : \text{Elem}] \\
\text{main} = \text{put} \rightarrow \text{get} \rightarrow \text{main} \\
\text{buf} : \text{seq} \rightarrow \text{get} \rightarrow \text{main} \\
\#\text{buf} \leq 1 \\
\Delta(\text{buf}) \\
i? : \text{Elem} \\
\text{buf}' = (i?) \smallsetminus \text{buf} \\
\Delta(\text{buf}) \\
o! : \text{Elem} \\
\text{buf} = \text{buf}' \smallsetminus \{o!\} \\
\end{array}
\]

\(^1\) We have added a state invariant to illustrate some of the results later in the paper.
Here, we have directly put the CSP part into the class schema, as is the correct syntax for CSP–OZ. The CSP process declarations are always placed directly after the interface description and before the state schema. We refer to the Object-Z part of this buffer specification containing interface, state schema, init schema as well as operation schemas as OZ\textsuperscript{Buf} and the CSP part as CSP\textsubscript{Buf}, and alternatively write Buffer as OZ\textsuperscript{Buf} \parallel \{\{put, get\}\} CSP\textsubscript{Buf}.

This model specifies one particular aspect, that this is a one-place buffer, of the system in two different ways: the state-based view defines a class invariant \#buf ≤ 1 and the dynamic view defines put and get to take turns.

We now transform the views to make this particular aspect more visible by splitting the views into this aspect plus a remainder. For the CSP part, we split main into the following two CSP processes

\[
\begin{align*}
\text{main}_1 = & \text{put} \rightarrow \text{main}_1 \quad \Box \quad \text{get} \rightarrow \text{main}_1 \\
\text{main}_{\text{Con}} = & \text{put} \rightarrow \text{get} \rightarrow \text{main}_{\text{Con}}
\end{align*}
\]

so that main \[=\] main\(_1\) \parallel \{\{\text{put}, \text{get}\}\} main_{\text{Con}}, where the set \{\{\text{put}, \text{get}\}\} used in the synchronisation is the joint alphabet of main\(_1\) and main\(_{\text{Con}}\).

Similarly, we decompose the Object-Z specification into two classes

<table>
<thead>
<tr>
<th>OZ</th>
<th>Init</th>
</tr>
</thead>
<tbody>
<tr>
<td>buf : seq Elem</td>
<td>buf = \langle \rangle</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>put</th>
<th>get</th>
</tr>
</thead>
<tbody>
<tr>
<td>[\Delta \text{buf}]\</td>
<td>[\Delta \text{buf}]\</td>
</tr>
<tr>
<td>i? : Elem</td>
<td>o! : Elem</td>
</tr>
<tr>
<td>buf' = \langle i? \rangle \bowtie buf</td>
<td>buf = buf' \bowtie \langle o! \rangle</td>
</tr>
</tbody>
</table>

and

<table>
<thead>
<tr>
<th>Con</th>
</tr>
</thead>
<tbody>
<tr>
<td>buf : seq Elem</td>
</tr>
<tr>
<td>#buf \leq 1</td>
</tr>
</tbody>
</table>

so that OZ\textsuperscript{Buf} can be expressed as follows:

\[
\begin{align*}
\text{OZ}_\text{Buf} \parallel & \quad \text{OZ} \\
\text{OZ} \quad \parallel & \quad \text{CSP}_\text{Buf} \\
\text{CSP}_\text{Buf} \quad \parallel & \quad (\text{main}_1 \parallel \text{main}_{\text{Con}}).
\end{align*}
\]

where such inheritance of classes is defined by conjunction of their state, initialisation and operation schemas [35]. In the following, we also write this simply as conjunction, i.e., OZ\textsuperscript{Buf} = OZ \land Con.

The two views now both contain (similar) constraints on the possible behaviour. However, since they impose the same kind of restriction on the base system (namely, that the buffer is one-place), it should be possible to have the restriction on just one view. Thus, we should be able to perform the transformation that adds a constraint to the dynamic view:

\[
\begin{align*}
\text{OZ}_\text{Buf} \parallel & \quad \text{main}_1 \\
\text{main}_1 \quad \parallel & \quad \text{main}_{\text{Con}}.
\end{align*}
\]

Here, and in the next section, we assume that the Object-Z and CSP parts have the same alphabet, and we use the parallel composition operator \parallel without synchronisation set to mean parallel composition with synchronisation on this joint alphabet. Section 4.3 discusses techniques for treating the case where CSP and Object-Z part have different alphabets. The above transformation is, in fact, only one of four possible types of transformations. We can add or remove a constraint from the Object-Z part while keeping the CSP part:

Case 1. Adding a constraint to the state-based view:

\[
\begin{align*}
\text{OZ} \quad \parallel & \quad \text{CSP}_\text{Buf} \\
\text{CSP}_\text{Buf} \quad \parallel & \quad (\text{OZ} \land \text{Con}) \quad \parallel \quad \text{CSP}_\text{Buf}.
\end{align*}
\]
Case 2. Removing a constraint from the state-based view:

\[ (OZ \wedge Con) \quad || \quad CSP_{Buf} \]

\[ \Downarrow \]

\[ OZ \quad || \quad CSP_{Buf} \].

Alternatively, we can keep the Object-Z part whilst adding or removing a constraint from the CSP part:

Case 3. Adding a constraint to the dynamic view:

\[ OZ_{Buf} \quad || \quad main_1 \]

\[ \Downarrow \]

\[ OZ_{Buf} \quad || \quad (main_1 || main_{Con}). \]

Case 4. Removing a constraint from the dynamic view:

\[ OZ_{Buf} \quad || \quad (main_1 || main_{Con}) \]

\[ \Downarrow \]

\[ OZ_{Buf} \quad || \quad main_1. \]

All these are valid behaviour-preserving transformations in the combined specifications. That is, the combined transformed specification is a refinement of the original combined description. However, if we look at the views in isolation we see that the transformations of the individual views are not behaviour-preserving: cases 1 and 2 are not data refinements on the Object-Z parts, and cases 3 and 4 not failures refinements on the CSP parts.

4.2. Transformations on static and dynamic views

We thus have to find conditions which guarantee behaviour preservation of such transformations, which can be stated in general terms and not restricted to the particular example presented. One possibility is to compute the transition systems of the complete specifications and compare these with respect to failures refinement. However, this is rather cumbersome and contradicts the idea of proving correctness of transformations on views. The intuitive argument for correctness so far has been ‘the restrictions imposed by a constraint are already covered by another view’. We now formalise this idea.

4.2.1. Static, or state-based, view

We start with transformations on the Object-Z part. Cases 1 and 2 above add or remove constraints, i.e., they can be seen as transformations involving a conjunction with some additional predicate in the class invariant (state schema), in the initialisation schema or in some operation schema. A conjunctive addition of such a predicate removes some of the behaviour of a class, and is sound if the CSP part, which also imposes restrictions on the allowed behaviour, already removes this behaviour anyway. It is this latter aspect that needs to be checked.

However, the CSP part is not able to impose restrictions on parameters of events if these parameters are determined by the values of variables of the Object-Z part. Thus, the CSP part can only cover the restrictions of a new Object-Z constraint if this constraint is data-independent. A data-independent constraint only imposes restrictions on the execution of operations, not the values of inputs and outputs. This is formalised in the following definition, where we use the notation \( \triangleright \) OP to strip off inputs and outputs from events, e.g., \( \langle \text{put.} 1, \text{get.} 2 \rangle \triangleright \) OP = \( \langle \text{put}, \text{get} \rangle \). The definition says that the relevant part of a trace is the ordering of operation names, and that these are not restricted by the constraint Con.

**Definition 7.** An Object-Z constraint Con is data-independent wrt. an Object-Z specification \( OZ_A \) with labelled transition system \( T_A = (Q_A, Q_0_A, E_A, \rightarrow_A) \) and \( T_{A\wedge Con} = (Q_{A\wedge Con}, Q_0_{A\wedge Con}, E_{A\wedge Con}, \rightarrow_{A\wedge Con}) \) iff the following holds:

\[ \forall \text{tr} : \text{traces}(T_A), z_0 : Q_0_A, z_n : Q_A \mid z_0 \Rightarrow_A z_n \bullet \]

\[ \text{tr} \triangleright \text{OP} \in \text{traces}(T_{A\wedge Con}) \mid \text{OP} \Rightarrow z_0 \Rightarrow_{A\wedge Con} z_n \sqcup \]

Furthermore, we require constraints to be deterministic, i.e., consistently rule out traces of behaviours.

**Definition 8.** An Object-Z constraint Con is deterministic wrt. an Object-Z specification \( OZ_A \) with labelled transition system \( T_A = (Q_A, Q_0_A, E_A, \rightarrow_A) \) and \( T_{A\wedge Con} = (Q_{A\wedge Con}, Q_0_{A\wedge Con}, E_{A\wedge Con}, \rightarrow_{A\wedge Con}) \) iff the following holds:

\[ \forall \text{tr} : \text{traces}(T_A), ev : \text{Event}, z_0 : Q_0_A, z_n : Q_A \bullet \]

\[ z_0 \Rightarrow_A z_n \wedge z_n \Rightarrow_{A\wedge Con} \]

\[ \Rightarrow (z_r \lor (ev)) \mid \text{OP} \notin \text{traces}(T_{A\wedge Con}) \mid \text{OP}. \]

A deterministic constraint thus always keeps or always rules out an event \( ev \) after a trace \( tr \). As we shall see in **Theorem 1**, if the constraint Con is data-independent and deterministic, we can check whether the CSP part already captures sufficient restrictions to ensure valid transformations on the traces only, by checking

\[ \text{traces}(CSP) \mid \text{OP} \subseteq \text{traces}(OZ_A \wedge Con) \mid \text{OP}. \]
Here, we again use projection onto event names to rule out differences in data values allowed by the CSP and Object-Z parts, which are irrelevant for this comparison. Note that the above trace condition is a comparison involving only single views, never their combination. However, in contrast to separate refinement checks, this is a comparison across views: the traces of the dynamic view are compared to the traces of the static view.

For the buffer (cases 1 and 2), we thus have to check whether the predicate #buf < 1 is data-independent. This is true since the predicate only refers to the size of the buffer not the actual values of elements in it. Furthermore, the constraint is deterministic: a trace tr completely determines the current length of the buffer and whenever an event ev is refused after a trace tr because of this constraint, tr \textbackslash (ev) will never be a valid trace. Furthermore, we have to check whether the trace condition holds, but a simple calculation shows that:

\[ \text{traces(main)} \models \text{OP} = \{e, (put), (put, get), \ldots \} = \text{traces(OZ}_A \land \text{Con}) \models \text{OP}. \]

We encapsulate these results in the following theorem, which will allow us to prove behaviour preservation for cases 1 and 2 above. Again, \(||\) stands for parallel composition with synchronisation on the joint alphabet of the CSP and Object-Z parts.

**Theorem 1.** Let A, C be CSP–OZ classes with common CSP part P_A, and Object-Z parts OZ_A, OZ_C, respectively. Let OZ_C be constructed from OZ_A by adding a constraint Con to the state, initialisation or operation schema. That is, OZ_C = OZ_A \land Con, so that A = OZ_A \parallel P_A and C = (OZ_A \land Con) \parallel P_A. Assume furthermore that the CSP and Object-Z parts have the same alphabet. If

1. Con is data-independent and deterministic wrt. OZ_A and
2. \text{traces}(P_A) \models \text{OP} \subseteq \text{traces}(OZ_A \land \text{Con}) \models \text{OP},

then A \sqsubseteq_f C and C \sqsubseteq_f A. That is, A =_f C.

**Proof**

We let OZ_A = (AState, Allit, (AOp_j)_{j \in J}), OZ_C = (CState, Clit, (COp_j)_{j \in J}), and let S be the joint alphabet of the Object-Z and CSP parts.

A \sqsubseteq_f C We have to show that failures(C) \subseteq failures(A). We prove this by contradiction. Let (tr, X) \in failures(A) but (tr, X) \notin failures(C), and let T_A = (Q_A, Q_0_A, E_A, \rightarrow_A) and T_C = (Q_C, Q_0_C, E_C, \rightarrow_C) be the transition systems of A and C, respectively. Furthermore, take \rightarrow (without index) to be the transitions defined by the operational semantics of CSP.

If (tr, X) \in failures(T_C), then there are states (z_0, P_A) \in Q_0, C and (z_n, P_n) \in Q_C such that (z_0, P_A) \xrightarrow{tr} C (z_n, P_n), (z_n, P_n) is stable in T_C and init_C (z_n, P_n) \cap X = \varnothing.

Since we have only added an invariant, every behaviour possible for OZ_C is also possible for OZ_A.

Hence, we also have (z_0, P_A) \xrightarrow{tr}_A (z_n, P_n). Since (tr, X) \notin failures(A) there must be a visible event Op.i.o \in X such that (z_n, P_n) \xrightarrow{Op.i.o}_A, but (z_n, P_n) \xrightarrow{Op.i.o}_C. By definition of the parallel composition of transition systems and since — assuming same alphabets — Op must be synchronised on, we get $z_n \xrightarrow{Op.i.o}_{OZ_A} P_n \xrightarrow{Op.i.o}_{OZ_A} A$ and $z_n \xrightarrow{Op.i.o}_{OZ_C} C.$

If (tr \land (Op.i.o)) \models \text{OP} \in \text{traces}(OZ_C) \models \text{OP} then by determinism of the constraint we cannot have $z_n \xrightarrow{Op.i.o}_{OZ_A}$ and $z_n \xrightarrow{Op.i.o}_{OZ_C}$. Hence, (tr \land (Op.i.o)) \models \text{OP} \notin \text{traces}(OZ_A) \models \text{OP}.

By (2) (tr \land (Op.i.o)) \models \text{OP} \notin \text{traces}(P_A) \models \text{OP} and hence P_n \xrightarrow{Op.i.o}_A which gives the contradiction.

C \sqsubseteq_f A We have to show that failures(A) \subseteq failures(C). We again prove this by contradiction. Let (tr, X) \in failures(A) but (tr, X) \notin failures(C), and let T_A = (Q_A, Q_0_A, E_A, \rightarrow_A) and T_C = (Q_C, Q_0_C, E_C, \rightarrow_C) be the transition systems of A and C, respectively.

If (tr, X) \in failures(T_A), then there are states (z_0, P_A) \in Q_0_A, (z_n, P_n) \in Q_A such that (z_0, P_A) \xrightarrow{tr}_A (z_n, P_n), (z_n, P_n) stable in T_A and init_A (z_n, P_n) \cap X = \varnothing. Hence, the trace tr is in particular also a trace of the CSP part alone: \text{tr} \models \text{OP} \in \text{traces}(P_A) \models \text{OP}. By condition (2), we thus get tr \models \text{OP} \in \text{traces}(OZ_A \land \text{Con}) \models \text{OP}. By data-independence of the constraint, we thus also have $z_0 \xrightarrow{tr}_C z_n$.

Now, assume there is a visible event Op.i.o \in X (i.e. (z_n, P_n) \xrightarrow{Op.i.o}_{OZ_A}) such that (z_n, P_n) \xrightarrow{Op.i.o}_C. Then, there are two possibilities: (i) $P_n \xrightarrow{Op.i.o}_A$, but then also $z_n \xrightarrow{Op.i.o}_{OZ_A}$, or (ii) $P_n \xrightarrow{Op.i.o}_{OZ_A}$, but then also $z_n \xrightarrow{Op.i.o}_{OZ_A}$, or $P_n \xrightarrow{Op.i.o}_{OZ_A}$, and thus $z_n \xrightarrow{Op.i.o}_{OZ_A}$, which contradicts $z_n \xrightarrow{Op.i.o}_{OZ_A}$.

Note first that this result implies behaviour preservation for both addition and removal of a constraint.

Second, note that we assumed that the CSP and Object-Z parts have the same alphabet. Only if the CSP part specifies restrictions on all of the Object-Z operations can we expect it to rule out the effect of a new constraint. Later on, we will see how to relax this constraint.
4.2.2. Dynamic view

For the addition and removal of constraints in the CSP part, we essentially take the same approach as we did for the state-based view. However, for this we first have to find the analogue of a conjunctive addition of a constraint in the CSP world. For Object-Z, a constraint just removes some of the behaviour. In CSP, this can be achieved by the use of parallel composition with synchronisation on all events in the constraint [38]. The process acting as a constraint then, like the Object-Z constraints, has to be deterministic and data-independent.

**Definition 9.** A CSP constraint $Con$ is data-independent if the following holds:

\[ \forall tr, tr' : \text{Events}^* \bullet \]

\[ tr | OP = tr' | OP \Rightarrow (tr \in \text{traces}(Con) \Leftrightarrow tr' \in \text{traces}(Con)). \]

A data-independent CSP process only fixes the orderings of operations, not of events.

**Definition 10.** A CSP constraint $Con$ is deterministic [33] if the following holds:

\[ \forall tr : \text{Events}^*, ev : \text{Events} \bullet \]

\[ tr ^\prec (ev) \in \text{traces}(Con) \Rightarrow (tr, \{ev\}) \notin \text{failures}(Con). \]

A deterministic process cannot both accept and refuse an event after a trace. For example, the process `main` in the buffer example is deterministic. FDR [13] can be used to check whether a CSP process is deterministic.

Note that the difference in the definition of deterministic and data-independent Object-Z and CSP constraints is due to the fact that a CSP constraint is a process and can thus stand on its own. An Object-Z constraint on the other hand may just be a predicate alone and is thus not a syntactically valid Object-Z specification. To be able to define restrictions on its behaviour, Object-Z constraints need to be combined with existing Object-Z specifications.

With this characterisation of deterministic and data-independent CSP constraints, we can treat cases 3 and 4.

**Theorem 2.** Let $A, C$ be CSP–OZ classes with common Object-Z part $OZ$ and CSP parts $P_A$ and $P_C$, respectively. Let $P_C$ be constructed from $P_A$ by parallel composition with a process $Con$, i.e., $P_C = P_A || Con$. Assume furthermore that $OZ$, $P_A$ and $Con$ all have the same alphabet.

If

1. $Con$ is deterministic and data-independent and
2. $\text{traces}(OZ) \cap OP \subseteq \text{traces}(P_A || Con) \cap OP$,

then $C \models_f A$ and $A \models_f C$. That is, $A \equiv_f C$.

**Proof**

Again, let $T_A = (Q_A, Q_{0,A}, E_A, \rightarrow_A)$ and $T_C = (Q_C, Q_{0,C}, E_C, \rightarrow_C)$ be the transition systems of $A$ and $C$, respectively. The proof proceeds by contradiction.

$A \models_f C$ Let $(tr, X) \in \text{failures}(C)$ but $(tr, X) \notin \text{failures}(A)$. Then, there are states $(z_0, P_A || Con) \in Q_{0,A}$ and $(z_n, P_n || Con_n) \in Q_C$ such that $(z_0, P_A || Con) \xrightarrow{Op.i.o} (z_n, P_n || Con_n)$. Hence, in particular $(z_0, P_A) \xrightarrow{A} (z_n, P_n)$. Assume again to have some $Op.i.o \in X$ such that $(z_n, P_n || Con_n) \xrightarrow{Op.i.o} (z_{n'}, P_{n'} || Con_{n'})$ but that $(z_n, P_n) \xrightarrow{Op.i.o} A$. Then $(tr ^\prec (Op.i.o)) \in \text{traces}(OZ)$, thus by (2) $(tr ^\prec (Op.i.o)) \xrightarrow{A} \notin \text{traces}(P_A || Con) \cap OP$. By data-independence, we get $(tr ^\prec (Op.i.o)) \notin \text{traces}(P_A || Con)$. Since $Con$ is deterministic, we — as a result — furthermore get $(tr, \{Op.i.o\}) \notin F(Con)$, thus $Con_n \xrightarrow{Op.i.o} A$, which contradicts the assumption.

$C \models_f A$ Let $(tr, X) \in \text{failures}(A)$, but $(tr, X) \notin \text{failures}(C)$. Then, there are states $(z_0, P_A) \in Q_{0,A}$ and $(z_n, P_n) \in Q_A$ such that $(z_0, P_A) \xrightarrow{A} (z_n, P_n)$ and there is some $Op.i.o \in X$ such that $(z_n, P_n) \xrightarrow{Op.i.o} A$. Thus, $(tr | OP) \in \text{traces}(OZ) \cap OP$ and by (2) $(tr | OP) \in \text{traces}(P_A || Con) \cap OP$. Hence, by data-independence of $Con$ it follows that there is also some $Con_n$ such that $(z_0, P_A || Con) \xrightarrow{Op.i.o} (z_n, P_n || Con_n)$. Since, however, $(z_n, P_n)$ refuses $Op.i.o$, so does $(z_n, P_n || Con_n)$ which contradicts the fact that $(tr, X) \notin \text{failures}(C)$. \( \square \)

Looking again at the buffer example, we noted already that the process `mainCon` which is defined as `put` $\rightarrow$ `get` $\rightarrow$ `mainCon` is deterministic. Furthermore, it is data-independent as it fixes no values for parameters. All that remains to be done is to check the trace condition. The traces of $OZ_{buf}$ are those of a one-place buffer, which are contained in the traces of $(\text{main}_1 || \text{mainCon})$ as required. Thus, transformations as described in cases 3 and 4 above can be verified for the buffer using this theorem.
4.3. Relaxing the constraint on identical alphabets

Theorems 1 and 2 require that the alphabets of the Object-Z and CSP components are the same. However, this assumption can be relaxed using the following idea. There are two cases: when the Object-Z component has events not in the alphabet of the CSP component; and when the CSP component has events not in the alphabet of the Object-Z component.

The technique in each case is similar. For the latter, we transform the Object-Z class $OZ_A$ to $OZ'_A$ by adding operations $Op \equiv [\text{true}]$ for each operation $Op$ in the alphabet of CSP but not in the alphabet of $OZ_A$. These new operations act as ‘skip’ events, not affecting the behaviour of $OZ_A$, the overall stable failures semantics of the combination of CSP and Object-Z remains the same. More formally, in the transition system of the Object-Z class all states of type $\text{Bind} \times OP \times \text{In}$ are augmented by tuples $(z, \text{Op}, i, z, o)$ for the new operation $Op$. Here, $i$ and $o$ can take arbitrary values allowed by the interface specification of $Op$. The main point is that the operation does not change the state (the binding $z$ remains the same). In the parallel composition with the CSP part, we have two cases to look at: transitions out of states $(z, q_2)$, where $z \in \text{Bind}$ and $q_2$ is a CSP term, and transitions out of states $(Z, q_2)$ where $Z \in (\text{Bind} \times OP \times \text{In}) \rightarrow (\text{Bind} \times Out)$.

1. A transition $(z, q_2) \xrightarrow{\text{ev}} (z, q'_2)$ (for $\text{ev}$ an event of such a new operation $Op$ of the Object-Z part) is replaced by a sequence of transitions $(z, q_2) \xrightarrow{\tau} (Z, q_2) \xrightarrow{\text{ev}} (z, q'_2)$. Here, the $\tau$-transition (and thus $Z$) follows the semantics for Object-Z as given in Definition 2. Since the state $(z, q_2)$ is thus not stable, the stable failures semantics of the overall transition system remains the same.

2. A transition $(Z, q_2) \xrightarrow{\text{ev}} (Z, q'_2)$ is replaced by a sequence $(Z', q_2) \xrightarrow{\text{ev}} (z, q'_2) \xrightarrow{\tau} (Z', q'_2)$, where $Z'$ is the augmentation of $Z$ with tuples for the new operation $Op$ (see above). Again, this does not affect the stable failures semantics.

Hence, the alphabet of $OZ'_A$ and the CSP component main can simply be made identical without changing the semantics, and Theorems 1 and 2 can be applied to this combination.

For the other direction, when the Object-Z component has events not in the alphabet of main, we transform main to main’ where:

$$\text{main'} = \text{main} \parallel PP = op_1 \rightarrow P \mathbin{\square} \cdots \mathbin{\square} op_n \rightarrow P$$

and $op_1, \ldots, op_n$ are the new operations. Here, the overall semantics is preserved as well: transitions $(q_1, Q) \xrightarrow{\text{ev}} (q'_1, Q)$ are replaced by transitions $(q_1, Q \parallel P) \xrightarrow{\text{ev}} (q'_1, Q \parallel P)$ which has no effect on the stable failures. The two alphabets are now identical and Theorems 1 and 2 can be applied to the new components.

5. Case studies

Finally, we take a look at some more complex examples, namely the HTS system discussed in Section 2 and a specification of the Java Vector class.

5.1. Holonic manufacturing system

We can apply the above results to show correctness of the transformations of the HTS system. The changes we have made are conjunctive changes of the Object-Z class, thus Theorem 1 is applicable, and we have to show data-independence, determinism and the traces condition.

In fact, there is a proof technique which can be used to show data-independence of a conjunctive change of an Object-Z class without computing the transition systems, namely by defining a simulation-like relation between the two classes. States are equivalent in this relation if they are reached by the same method calls (regardless of i/o-parameters). One then has to check that on related states, $\text{Con}$ only blocks method calls due to their name, and not their i/o-parameters. For example, in the HTS, states are related if offers has the same size: the size of the sequence in a particular state does not depend on parameters of previous offers (and also not on order and choose), it just depends on the number of method executions. The addition of $\text{Con} = \#\text{offers} \leq 3$ then means that the ability to do an offer, order or choose method does not depend on the parameters of the methods, but just on which method is attempted. That is, if one offer.h.c is blocked, they all are.

In fact, for our case study we can check this condition syntactically: with the methods given, every predicate that only refers to the size of a sequence is data-independent and furthermore deterministic (for instance this does not apply to sets\footnote{1}).

Thus, the new constraint $\#\text{offers} \leq 3$ is data-independent and deterministic.

The remaining condition to be checked is the one that determines whether the CSP part sufficiently restricts the behaviour of the class so that the constraint does not introduce new behaviour. That is:

$$\text{traces(\text{main})} \parallel OP \subseteq \text{traces}(OZ_A \land \text{Con}) \parallel OP$$

A tractable way of checking this condition is by actually carrying out the translation of the Object-Z class to CSP, and afterwards using the CSP model checker FDR [13] to check for trace inclusion. The translation of Object-Z to CSP follows
an approach outlined in [16]. In [15], this translation was shown to be sound with respect to the failures semantics of CSP–OZ. Thus, we can safely use it to show properties of our CSP and Object-Z parts. An Object-Z class is translated into a CSP process (e.g., Machine2 becomes: process OZ(offers,orderTo), given below) parameterised with the variables of the class. The behaviour of this process is a choice (in CSP $[\{}$) over all possible method executions followed by a recursive call to the process possibly with modified instantiation of variables. The precondition of a method acts as a guard to the method execution (denoted &). For the definition of data types (HTS) we follow the Object-Z definitions. This generic process is then instantiated with initial values as given by the Object-Z specification. The check for trace inclusion can be performed by FDR provided the state spaces of both processes (the protocol and the translated Object-Z class) are finite. To this end we fix the set of Cost to \{1..5\}.

Below, the CSP specification of both the protocol and the Object-Z class are given in the syntax of the FDR model checker (CSP\(_M\)).

```plaintext
-- Declaration of data types
datatype HTS = h1 | h2 | h3
Hts = \{h1,h2,h3\}
Cost = \{1..5\}
-- Declaration of channels
channel offer : Hts.Cost
channel order : Hts
channel choose
channel offerProj -- channel offer without parameters
channel orderProj -- channel order without parameters
-- CSP process of protocol
main = FindHts; main
FindHts = ([|] h : Hts @ offer.h?x -> SKIP);(choose -> order?x -> SKIP)
-- CSP process of class Machine2
Machine2 = OZ(<>,h1)
OZ(offers,orderTo) =
  -- external choice over all methods of form
  -- precondition & method execution -> recursive call
  (offers != <> & let
   ot = first(min(offers))
   within choose -> OZ(offers,ot))
  [] (length(offers) < 3 & offer?h?c -> OZ(offers^<(h,c)>,orderTo))
  [] (order!orderTo -> OZ(<>,orderTo))
-- Projections of main and Machine2 to method names (by renaming)
mainProj =
  main[offer.h.c<-offerProj,order.h<-orderProj|h<-Hts,c<-Cost]  
Machine2Proj =
  Machine2[offer.h.c<-offerProj,order.h<-orderProj|h<-Hts,c<-Cost]
Projection (i.e., taking $[\{}$) is defined by renaming all channels to ones without parameters. The check for trace inclusion can be carried out by asking FDR to verify the assertion:

```
assert Machine2Proj $|$= mainProj
```
This returns a positive response, and thus both conditions in Theorem 1 hold and the overall transformation is valid for the particular choice of data types. Of course, the latter have to be fixed (and finite) for FDR to be able to make an exploration of the state space, and exploring the system for differing instantiations increases the confidence that the transformations are valid for arbitrary datatypes.

5.2. Java's Vector class

The second case study concerns the Object-Z specification of (part of) the Java Vector class. A Java vector can store an arbitrary number of objects. It implements the Java Enumeration interface, which allows for an enumeration of the elements in the vector. This is achieved using the methods elements to initialise the enumeration, hasMoreElements and notHasMoreElements to test for more elements, and nextElement to get the next object in the vector. Clearly, this is an abstract specification; it does not resemble the actual implementation of Vector, in particular, Vector does not guard against improper use.

The Object-Z specification only allows for the correct use: first, method elements is called, then a successive testing for more elements and getting an element is carried out until there are no more elements. We assume a given type Object, and,
in addition, to the operations specified below, we assume that the class has methods for storing and getting elements, etc., but these are elided.

```plaintext
Vector

method elements, hasMoreElements, notHasMoreElements
method nextElement : [o! : Object]

```

Here, the correct ordering of method invocation during an enumeration is achieved by two boolean variables, `enumStarted` and `moreElements`. To make this ordering explicit in the specification, we add a CSP process as follows:

```
Vector2

main = elements -> Enum
Enum = hasMoreElements -> nextElement -> Enum
\[ notHasMoreElements \rightarrow \text{main} \]

```

This is clearly not a refinement of the CSP view alone, since an empty process (i.e., `SKIP^3`) has been replaced by a process with many operations. However, in combination with the Object-Z part the transformation is valid. This is a change to the dynamic side, so we apply Theorem 2, noting the CSP process we are adding conjunctively is the process `main` in `Vector2`. Process \( P_a \) of Theorem 2 is empty, i.e. equal to `SKIP`. To check the two conditions of Theorem 2, we again use FDR. The first condition requires process `main` from `Vector2` to be deterministic. Omitting channel definitions this time, we get the following CSP\(_M\) code:

```
main = elements -> Enum
Enum = hasMoreElements -> nextElement?x -> Enum
\[ notHasMoreElements \rightarrow \text{main} \]
```

FDR’s check for determinism returns a positive response. Furthermore, the process is data-independent as it fixes the values for parameters. Next, we need to check the trace inclusion condition, i.e., check whether

```
traces(Vector) \mid OP \subseteq traces(SKIP \mid main) \mid OP
```

This would essentially show that the ordering of the Object-Z part as determined by the boolean variables is consistent with the traces of `main`. Omitting the definitions of datatypes this time, translating the Object-Z part of `Vector` to CSP gives:

---

\(^3\) To meet the conditions of our theorems about equality of alphabets, we would in principle need to start not with `SKIP` but with the chaotic CSP process allowing for all orderings. To simplify matters, we have chosen `SKIP` here. The overall result is, however, not affected.
OZ(elementData, elementCount, enumIndex, enumStarted, moreElements) =
  (not enumStarted & elements ->
   OZ(elementData, elementCount, 0, true, moreElements))
[] (enumStarted and moreElements &
   nextElement!nth(elementData, enumIndex + 1) ->
   OZ(elementData, elementCount, enumIndex + 1, enumStarted, false))
[] (enumStarted and (enumIndex < elementCount) and
   not moreElements &
   hasMoreElements ->
   OZ(elementData, elementCount, enumIndex, enumStarted, true))
[] (enumStarted and (enumIndex >= elementCount) and
   not moreElements &
   notHasMoreElements ->
   OZ(elementData, elementCount, enumIndex, false, false))

The translation again follows the principle outlined above for Machine2. We use a CSP process parameterised by the state variables of Vector. This process is defined as an external choice over all possible method executions. Every such execution is prefixed by a guard (unless the precondition of the method is true) and followed by a recursive call to the same CSP process changing the values of state variables as specified in the method.

The above check for trace inclusion as carried out by FDR again yields a positive response.

Next, we perform a second transformation of the specification and remove variables enumStarted and moreElements as well as all predicates referring to them from the Object-Z class. This transformation falls into the category of removing an Object-Z constraint. Again, this transformation is clearly not a valid refinement of the Object-Z part alone, and here we need to apply Theorem 1. The required check on the traces is similar and thus one just needs to check that the conjunctive addition of the predicates is data-independent and deterministic. That is, the predicates enumStarted, moreElements′, ¬enumStarted′ ∧ ¬moreElements′, etc., are all data-independent. In a fashion similar to the buffer specification, since these predicates are independent of the data in, e.g., elementData, they are all data-independent as required. Furthermore, the predicates are deterministic.

Thus, the final specification is the following.

Vector 3

method elements, hasMoreElements, notHasMoreElements
method nextElement : [o! : Object]
main = elements → Enum
Enum = hasMoreElements → nextElement → Enum
   □ notHasMoreElements → main

Init
enumIndex = 0
#elementData = elementCount

elementData : seq Object
elementCount : N
enumIndex : N

nextElement

Δ(enumIndex)
enumIndex′ = 0

hasMoreElements

enumIndex < elementCount

notHasMoreElements

enumIndex ≥ elementCount

The transformation into this specification is a valid refinement step. The final specification exhibits a neat separation into a protocol describing possible orderings of method executions and a data part. This, in particular enhances the readability of the specification.

6. Conclusion

The aim of this paper has been to derive techniques, whereby model transformations can be verified even if the sub-transformations on the individual views are not behaviour-preserving. We have placed this work in the context of a
combination of Object-Z and CSP, however, it should be clear that the techniques we have discussed could be transferable to other integrations. For example, we could have used a simple state machine for describing the protocol in our examples, and translated it to CSP in order to verify the model transformation later. Such a translation is employed in [31]. In particular, the work is applicable to (parts of) UML models, for example, when written in the profile proposed in [25].

The work presented here is an extension of [12], which only considered the conjunctive addition of constraints to the state-based view. Here, we additionally investigated the removal of constraints as well as the addition and removal of processes on the dynamic view, and this can be combined with changes due to data and process refinements. This enables more complex model transformations to be verified if they can be shown to be composed out of smaller, correct transformations.

Related work

Of course, there is a broad field of existing work on refinement and model transformations. For an overview of model transformations and refactorings on software see, for instance, [28,24]. Our work might best be classified using the following two criteria. The first is the number of views treated. There are a large number of approaches to refinement/refactoring/model transformations treating a single view only (or multiple views, but separately), e.g., [27,36,22,4,21] using for instance Object-Z specifications or various types of UML diagrams as basis, and all classical definitions of refinement. However, there are only a small number of approaches treating multiple views together [5,11,37].

A second criterion for classification are the techniques employed. Whilst a lot of work has been carried out on defining model transformations, most with the ultimate goal of automating them (e.g., the work on ATL [18]), others have been concerned with a posteriori verifying model transformations [32]. The first category is sometimes also called the operational and the second one the relational approach. According to this classification, our work falls into the category verifying multiple view transformations.

Our approach uses a standard notion of refinement as a correctness criterion for model transformations. The use of refinement concepts can, for instance, also be found in [27] (Object-Z class refinement), [4] (refinement between graph transformation systems) and [37] (CSP process refinement). Transformations involving more than one view are treated in [5], where refactorings for class diagrams and (simple) consecutive modifications on state machines and sequence diagrams are defined. Relational approaches, i.e., validations of transformations, involving more than one view can be found in [11] where Object-Z/CSP classes are split, and in the work of Schneider and Treharne (e.g., [37]). The latter have considered the refinement of an integration of CSP and B, and have discussed sufficient conditions for when the structure of a system changes. They, for example, isolate conditions by which refinements can be checked which are not compositional. This is similar to our motivation, however, the work of [37] is set in a general context, whereas our aim was to exploit the consequences of a particular situation.

Our work is also related to that on refinement in integrated formal notations, of which [37] is also an example. This strand of work includes that on structural refinement in another integration of Object-Z and CSP [10,11], as well as the integrations considered by Treharne and Schneider mentioned above. Other work in this vein includes TCOZ [23] and Circus [29] – an integration of Z, CSP and the refinement calculus. There has also been work on integrating formal notions with informal ones, for example, an integration of CSP–OZ with UML can be found in [26].

Future work

Further work in this area includes consideration of situations where we have simultaneous changes in the CSP part, whereby new events are added, and when new operations are added in the Object-Z part. We are, furthermore, interested in developing transformation patterns which, by construction, guarantee behaviour preservation. In fact, in the conditions isolated above this is already partly the case since we had a syntactic check for one of the conditions, and one would hope that further work in this area would determine some more relevant patterns of this type.

References