Failure-Divergence Semantics as a Formal Basis for an Object-Oriented Integrated Formal Method

Clemens Fischer and Heike Wehrheim
Universität Oldenburg
Fachbereich Informatik
Postfach 2503, D–26111 Oldenburg, Germany
{fischer,wehrheim}@informatik.uni-oldenburg.de

Abstract
The integration of several different modelling techniques into a single formal method has turned out to be advantageous in the formal design of software systems. Giving a semantics to an integrated formal method is currently a very active area of research. In this paper we discuss the advantages of a failure-divergence semantics for data and process integrating formal methods, in particular for those with a concept of inheritance. The discussion proceeds along the lines of the formal method CSP-OZ, a combination of CSP and Object-Z, developed in Oldenburg.

1 Introduction
Recently, there is an emerging interest in integrated specification methods combining different formalisms into one. The advantage of integrating several techniques into one is that it allows for more convenient specifications, choosing the most suitable technique for every aspect of the system. The most prominent representative for an integrated (although not completely formal) method is certainly the Unified Modelling Language UML [Obj99, BRJ99] which combines standard object-oriented class diagrams with a number of diagrams for describing the dynamic behaviour of a system. On the side of formal methods, combinations of process algebras with Z [TA97, Smi97, GS97], B [But99], abstract data types [BB87] or functional languages [GH96] or combinations of Z with statecharts [GHD98] can be found as examples for the integration of different formalisms.

This paper continues a series of previous articles on integration modelling in the Formal Specification Column. Differing from previous articles of the Column we will not propose an abstract integration paradigm or a metamodel here but discuss the semantic basis for one particular object-oriented integrated formal method, namely CSP-OZ [Fis97, Fis00]. CSP-OZ is a formal method which combines the state-oriented method Object-Z [DRS95, Smi00], an object-oriented extension of Z,
with the process algebra CSP [Ros97]. The semantic basis for this combination is the failure-divergence model of CSP. In this paper, we explain the design choices made when giving a failure-divergence semantics to CSP-OZ, and discuss the main advantages of this particular semantics.

CSP-OZ is only one of a number of combinations of Z with a process algebra. For a comparison of the different approaches see [Fis98].

2 CSP-OZ

CSP-OZ\(^1\) is an object-oriented formal method which integrates Object-Z and the process algebra CSP. Object-Z or in general Z is a state-oriented method and thus good at describing data types and operations. Z is currently being standardised by the ISO [Z99] and has been successfully applied in a number of industrial case studies. A large range of tools (for instance editors, parsers and theorem provers) support the writing of Z specifications. The process algebra CSP on the other hand is mainly concerned with the description of the dynamic behaviour of distributed communicating systems. For CSP a commercial model checker (FDR) can be used to perform correctness checks.

The basic idea behind the integration of these two formalisms is to identify every Object-Z class with a CSP process. For this, every class specification of Object-Z is equipped with a communication interface and a behaviour description in the style of CSP\(^2\). Then while the Object-Z part defines the attributes and methods of a class (possibly with a guard and an effect predicate for every method), the CSP part fixes the possible orderings of method execution. A method invocation is thus identified with a CSP communication event.

A CSP-OZ specification typically consists of a number of Z type definitions and CSP-OZ classes (possibly using inheritance as a structuring concept) together with a description of the architecture of the system, consisting of an instantiation of classes into objects and their composition via CSP operators.

CSP-OZ fits perfectly into the integration paradigm proposed in [EPO99], which consists of four hierarchically organised layers: in CSP-OZ layer 1 (describing data types) contains all Z axiomatic descriptions, data type definitions etc.; layer 2 (data states and transformations) contains pure Object-Z class descriptions, defining the state schema (attributes) of classes and the method schemas; layer 3 (processes) is captured by the CSP behaviour definitions within a class and layer 4 (architecture) is the final instantiation of classes into objects and their composition via CSP operators, most often parallel composition and hiding. The semantics of layer 1 is the standard Z semantics, the semantics of layer 2 lifts this basis to the failure-divergence model, and the upper level semantics are obtained by applying standard CSP theory.

\(^1\)http://semantik.Informatik.Uni-Oldenburg.DE/~csp-oz

\(^2\)To be precise, a CSP-OZ specification may also still contain pure Object-Z classes.
2.1 Example.

We explain this concept with the example of a vending machine.

Layer 1. The data types of the vending machine are given as Z types.

\[
[MONEY] \\
DRINK ::= TEA | COFFEE
\]

A drink can either be tea or coffee; \(MONEY\) is an uninterpreted basic type. We assume the existence of two functions and the relation \(\leq\) on \(MONEY\).

\[
\begin{align*}
price & : DRINK \rightarrow MONEY \\
\text{diff} & : MONEY \times MONEY \rightarrow MONEY \\
\leq & : MONEY \leftrightarrow MONEY
\end{align*}
\]

The function \(price\) gives the prices of a drink; \(\text{diff}\) computes the difference between two amounts of money.

Layer 2. For the second layer, we use the Object-Z class \(VM-Data\) as specified in Figure 1. It comprises five state variables (attributes): \(store\), a store for drinks; \(s\)-\(money\), a store for money; \(inserted\), an intermediate store for the inserted money; \(drink\), for the chosen drink; and the boolean flag \(ok\), to check whether enough money has been inserted.

Four operation schemas define the transformations on this state space (the methods of the class): \(\text{choose}\) stores a chosen drink in the variable \(drink\). The so called delta-list \(\Delta(drink)\) specifies that \(\text{choose}\) can only change the variable \(drink\). The schema \(\text{choose}\) has two parts: The declaration part above the line specifies the delta list and the parameter \(d\) of the operation. The predicate part below the line is a predicate describing the state transformation. The final value of \(drink\) is there written as \(\Delta(drink)\). The operation \(\text{pay}\) stores the inserted money; \(\text{change}\) calculates the change according to the chosen drink. If not enough money was inserted, \(ok\) is set to \(false\). Finally, the operation \(serve\) offers the chosen drink and decreases the value of the corresponding store.

Layer 3. For the third layer, we use a so called CSP-OZ class. It is an Object-Z class enriched with an interface and a CSP process. The interface is a list of typed channels. The CSP process restricts the behaviour of the class: First a drink must be chosen, then money has to be inserted, the change is given back and finally the drink is served. Because we have specified the possibility of inserting not enough money, serving the drink can be skipped (\(\sqcup\) denotes the choice operator). However, this is not controlled by the CSP part, rather we use Z schemas for this purpose, since this part depends on certain attribute values.

The class \(VM-Contrl\) (see Fig. 2) is a specialisation of the \(VM-Data\), inheriting all its attributes and methods. That makes all items declared in \(VM-Data\) available for \(VM-Contrl\).
<table>
<thead>
<tr>
<th>VM-Data</th>
<th>serve</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{ok, store} )</td>
<td>( \text{ok, store} )</td>
</tr>
<tr>
<td>( \text{ok, DRINK} )</td>
<td>( \text{ok, DRINK} )</td>
</tr>
<tr>
<td>( \text{d! : DRINK} )</td>
<td>( \text{d! : DRINK} )</td>
</tr>
<tr>
<td>( \text{d! = drink} \land \text{ok = true} )</td>
<td>( \text{d! = drink} \land \text{ok = true} )</td>
</tr>
<tr>
<td>( \text{store'}(\text{d!}) = \text{store}(\text{d!}) - 1 )</td>
<td>( \text{store'}(\text{d!}) = \text{store}(\text{d!}) - 1 )</td>
</tr>
<tr>
<td>( {\text{d!}} \leftarrow \text{store'} = {\text{d!}} \leftarrow \text{store} )</td>
<td>( {\text{d!}} \leftarrow \text{store'} = {\text{d!}} \leftarrow \text{store} )</td>
</tr>
<tr>
<td>( \text{ok!} = \text{false} )</td>
<td>( \text{ok!} = \text{false} )</td>
</tr>
</tbody>
</table>

| Figure 1: Layer 2: The Object-Z Class VM-Data |

To link Z operation schemas to behaviour, special keywords are used: `enable` specifies the guard of an operation, `effect` specifies the corresponding state transition. Thus a drink can only be chosen when it is stored and a drink is only served if the customer has paid enough money. The effect of all operations is mapped directly to the corresponding Object-Z operations. The schema `Init` specifies the allowed initial states.

In this example we have a very strict separation of data and behaviour aspects: the Object-Z class specifies the effect of operations on the data and the CSP-OZ class specifies all aspects relevant to the ordering of operation execution. This separation may also be weakened: the specification of data can be put directly into the CSP-OZ class, completely omitting an Object-Z class specification. For larger systems, it may however be useful to adhere to a strict separation in order to enhance modularity.
of the specification.

Layer 4. In layer 4 we combine CSP-OZ classes by standard CSP operators. An overview of the system is given in Fig. 3. There the class \textsc{VM-Contrl} communicates with four other classes which we have not explicitly given here: \textsc{Buttons} controls the two buttons to choose a drink, \textsc{Dispenser} controls the process of making coffee or tea, \textsc{Slot} stores inserted coins and \textsc{Change} hands out the change. The diagram in Figure 3 corresponds to the following CSP process:

\[
\text{VendingMachine} = (\text{Buttons} || \text{Dispenser} || \text{Change} || \text{Slot})
\]

where $||$ denotes interleaving, $\parallel$ parallel composition synchronising on the set of events $A$, $\not\!A$ hides the events from $A$ and $\{\text{choose, serve, change, pay}\}$ is the set of events corresponding to the channels \text{choose}, \text{serve}, \text{change} and \text{pay}.
2.2 Semantics

CSP-OZ possesses a uniform formal semantics on the basis of CSP’s failure-divergence model. A failure-divergence model records the possible execution traces of a system, the sets of events which may be refused after a particular trace and the points of divergence, i.e. execution of an infinite number of internal events (livelock). The semantics of a CSP-OZ class specification is derived in two steps:

- the Z part of the class is given a failure-divergence semantics (via a translation into CSP) and
- the semantics of the Z part is combined with the (standard) semantics of the CSP part via the CSP parallel composition operator.

The CSP-OZ semantics is built on the Z semantics as defined in the current committee draft of the ISO [Z99]. Types, expressions and schemas are interpreted according to this semantics. Every Object-Z class is then viewed as a process where the class’ methods play the role of communication events. The following four points explain the main considerations underlying the semantics of Object-Z classes:

- Every method whose guard evaluates to true in the current state can be executed,
• the values for input parameters are fixed by the environment communicating with the object (external choice over all possible values for input parameters),
• the values for output parameters and next state are internally chosen from the set of possible values according to the effect schema of the method (internal choice),
• an invocation of a method, whose guard evaluates to true but whose precondition of the effect predicate is not fulfilled leads to divergence.

This semantics supports a blocking as well as a non-blocking view of methods: guards can be used to block the execution of methods in particular states of the object, the execution of non-blocked methods whose effect predicates can however not be fulfilled leads to divergence (every behaviour possible after invocation of method). The non-blocking view has its origin in Z, where undefinedness is usually interpreted as ‘all possible values’. The blocking view is typically encountered in the design of distributed communicating systems, for which CSP can be used as a modelling language.

3 Failure-Divergence Semantics

The reference model of [GR99] for integrated formalisms proposes to use abstract transition systems as the semantical basis for layer 2 specifications. Although this is also possible for CSP-OZ [Fis00] (the transition system can then be the basis for deriving failures and divergences), we will directly discuss the failure-divergence semantics here. The choice for a failure-divergence semantics has several advantages.

▷ The failure-divergence model is the basis for defining three different implementation relations for CSP: trace refinement, stable failures refinement and failure-divergence refinement. These relations compare an implementation towards a given specification according to the dynamic behaviour they exhibit (process refinement), requiring that the implementation exhibits only a subset of the traces/failures/divergences of the specification. Having given a failure-divergence semantics to CSP-OZ, all these implementation relations are applicable to CSP-OZ specifications.

▷ All process refinement relations are monotone wrt. the operators on the architectural level (layer 4), which is clearly a prerequisite for a good implementation relation. By monotonicity we mean that the process refinement relations are preserved under all CSP operators. This is a standard result for CSP.

▷ Not only CSP but also Z comes with a specific refinement concept. The standard refinement concept for state-oriented formalisms is data refinement, used for instance for Z or VDM specifications. Data refinement is concerned with the change of data structures on different abstraction levels. An initial specification often uses different data structures than the final implementation which in its choice of data structures has to be oriented towards efficiency. Data refinement rules are used to show the correctness of the implementation with respect to the specification.
Since we have a clear separation of the Z- and CSP-part within a CSP-OZ class, the concept of data refinement can also be applied to CSP-OZ classes, i.e. it can be determined whether a class is a data refinement of another class. It is hence most important that this layer 2 (data) refinement concept neatly fits to the layer 3 (process) refinement concept. The relationship between data refinement and failures refinement has been intensively studied, and the duality of data and process refinement has been shown in [Jif89, dRE98]: given a failure-divergence semantics for a state-oriented formalism, data refinement induces process refinement. In [FH97] these results have been extended to CSP-OZ with its specific combination of blocking and non-blocking views of method invocation.

CSP-OZ is an object-oriented formal method. Specifications define a number of classes which may be instantiated into objects and afterwards combined. Classes may inherit attributes, methods or behaviour of superclasses. Semantically inheritance is interpreted as the conjunction of the Z-part and the parallel composition of the CSP-part of a class. As usual, inheritance in CSP-OZ is primarily a concept for re-use of already written specification code. However, closely connected with inheritance is the question of subtyping: when is a subclass a proper subtype of its superclass? Subtypes always have to fulfil the principle of type substitutability: an exchange of a superclass by a subclass should be transparent to clients. In the context of process-oriented formalisms, subtyping is not so much concerned with types in the usual sense but with the behaviour associated with a class. A replacement of a superclass by one of its subclasses is only possible if the subclass closely resembles the behaviour of the superclass. One candidate for such a behavioural subtyping relation obviously seems to be failure-divergence refinement. Failure-divergence refinement is however not able to capture the issue of extension of functionality (i.e. addition of new methods to the subclass) which arises when using inheritance. Nevertheless failure-divergence semantics can be the basis for defining a number of behavioural subtyping relations, varying in the degree in which they fulfil the principle of type substitutability [FW00]. The principle idea is to use various kinds of restriction and hiding operators of process algebras (applied to the methods added in the subclass) when comparing superclasses with subclasses. In [FW00] the adequateness of such behavioural subtyping relations has been shown by giving testing characterisations for the relations.

Summarising, failure-divergence semantics provides a clear basis for formal methods integrating state and process based languages. The refinement concepts of the different layers fit well together and the composition operators of the architectural level preserve the refinement relations. Moreover, a failure-divergence model can also be used for defining behavioural subtyping relations; a failure-divergence semantics for an object-oriented formal method thus additionally gives us the possibility of determining subtype relationships among classes.
References


