Available online at www.sciencedirect.com

SCIENCE DIRECT®

Science of Computer Programming 55 (2005) 227–257

Science of Computer Programming

www.elsevier.com/locate/scico

Specification and (property) inheritance in CSP-OZ☆

Ernst-Rüdiger Olderog*, Heike Wehrheim

Department of Computing Science, University of Oldenburg, 26111 Oldenburg, Germany

Received 31 August 2003; received in revised form 15 April 2004; accepted 30 May 2004

Abstract

CSP-OZ [C. Fischer, CSP-OZ: A combination of Object-Z and CSP, in: H. Bowman, J. Derrick (Eds.), Formal Methods for Open Object-Based Distributed Systems, FMOODS'97, vol. 2, Chapman & Hall, 1997, pp. 423–438] is a combination of Communicating Sequential Processes (CSP) and Object-Z (OZ). It enables the specification of systems having both a state-based and a behaviour-oriented view using the object-oriented concepts of classes, instantiation and inheritance. CSP-OZ has a process semantics in the failure divergence model of CSP. In this paper we explain CSP-OZ and investigate the notion of inheritance. Furthermore, we study the issue of property inheritance among classes. We prove in a uniform way that behavioural subtyping relations between classes introduced in [H. Wehrheim, Behavioural subtyping in object-oriented specification formalisms, University of Oldenburg, Habilitation Thesis, 2002] guarantee the inheritance of safety and "liveness" properties.

Keywords: CSP; Object-Z; Failures divergence semantics; Inheritance; Safety and "liveness" properties; Model-checking; FDR

☆ This research was partially supported by the DFG under grant Ol/98-3.

* Corresponding author.

E-mail addresses: olderog@informatik.uni-oldenburg.de (E.-R. Olderog), wehrheim@informatik.uni-oldenburg.de (H. Wehrheim).

0167-6423/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2004.05.017

1. Introduction

In contrast to the widespread use of object-oriented programming and specification languages, little is known about the properties enjoyed by systems constructed in the object-oriented style. Research on verification of object-oriented descriptions often takes place in the setting of object-oriented programming languages, for instance Java. The methods range from Hoare-style verification supported by theorem provers [22,32] via static checkers [26] to model-checking techniques [18]. Verification of object-oriented *modeling* languages focuses on UML (e.g. [24,36]). These approaches check properties of UML state machines by translating them into existing model-checkers. Although UML is an integrated formalism allowing the specification of data and behaviour aspects, existing model-checking techniques focus on the behavioural view.

Reasoning about object-oriented specifications represents a challenge in its own right. Given a proof made with respect to one data-model, its reuse in another data-model extended by inheritance represents a major problem that must be overcome in order to build up libraries that support proofs about non-trivial applications. To describe the preservation of behavioural properties of classes under change the concept of subtyping has been lifted from data types to objects by [1,27]. Subtyping definitions used in the verification of object-oriented *programming languages* (like Java) are mainly variations of this basic idea [32,8]. Whereas these approaches are restricted to state-based specifications (using e.g. Object-Z [39], Larch [17]) and consequently to state-based properties like class invariants, Nierstrasz [29] proposed definitions suitable to deal with behaviour-oriented specifications (using e.g. CSP [20]). A first systematic study of subtyping for specifications integrating state-based and behaviour-oriented views is [45].

In this paper we study specification and (property) inheritance in the language CSP-OZ [10,12] combining two existing specification languages for processes and data. In particular, we develop two subtyping relations for CSP-OZ which guarantee preservation of safety and (a form of) liveness properties.

Specification of processes. *Communicating Sequential Processes* (CSP) was introduced by Hoare [19,20]. The central concepts of CSP are synchronous communication via channels between different processes, parallel composition and hiding of internal communication. For CSP a rich mathematical theory comprising operational, denotational and algebraic semantics with consistency proofs has been developed [3,30,34]. Tool support comes through the FDR model-checker [33]. The name stands for Failure Divergence Refinement and refers to the standard semantic model of CSP, the failure divergence model $\mathcal{FD}$, and its notion of process refinement. Besides $\mathcal{FD}$ there is also the simplified trace model $\mathcal{T}$ of CSP. In $\mathcal{T}$ process refinement can be used to model safety properties via trace set inclusion, and in $\mathcal{FD}$ process refinement can be used to model a limited form of "liveness" properties. Here "liveness" refers to the absence of divergence, i.e. infinite internal activity that prevents a process from ever reacting to external requests of the environment.

Specification of data. *Z* was introduced in the early 1980's in Oxford by Abrial as a set-theoretic and predicate language for the specification of data, state spaces and state transformations. It comprises the mathematical tool kit, a collection of convenient notations

and definitions, and the schema calculus for structuring large state spaces and their transformations. A Z schema has a name, say S, and consists of variable declarations and a predicate constraining the values of these variables. It is denoted as follows:

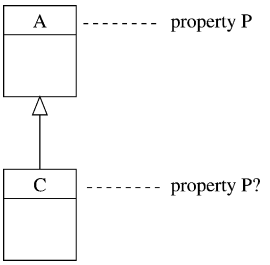
S

declarations

predicate

The first systematic description of Z is [42]. Since then the language has been published extensively (e.g. [48]) and used in many case studies and industrial projects. *Object-Z* is an object-oriented extension of Z [9,37,39]. It comprises the concepts of classes, instantiation and inheritance. Z and Object-Z come with the concept of data refinement. For Z there exist proof systems for establishing properties of specifications and refinements such as Z/EVES [35] or HOL-Z based on Isabelle [23]. For Object-Z type checkers exist. Verification support is less developed except for an extension of HOL-Z [41].

Property inheritance. Inheritance allows the re-use of existing specifications and code. Besides the re-use of specifications the re-use of *correctness proofs* is an important issue in any formal approach to system development. Given a class *A* with some proven property *P* we are interested in conditions under which this property is inherited to specialised classes *C*.



For this purpose, a notion of subtyping for the combined formalism CSP-OZ is developed and the inheritance of safety and "liveness" properties is studied.

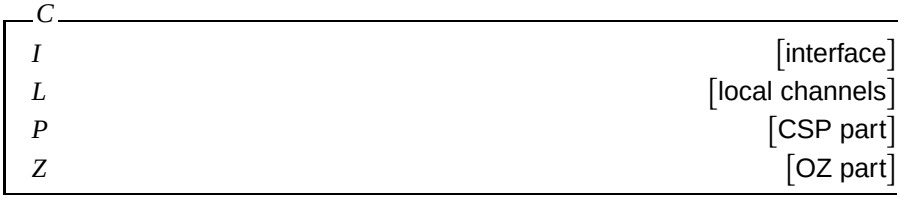
Structure of this paper. Section 2 introduces the combination CSP-OZ by way of an example. In Section 3 the semantics of CSP-OZ is reviewed. Section 4 is devoted to the inheritance operator in CSP-OZ and its semantics. In Section 5 the inheritance of properties is studied in depth, and the relationship between inheritance and subtyping is discussed. Proofs of the new theorems are given in Section 6. Finally, we conclude with Section 7.

A preliminary version of this article appeared as [31]. This journal version contains more details of the running example and the semantics, a new section on inheritance and subtyping, and proofs of the theorems on property inheritance.

2. The combination CSP-OZ

There are various specialised techniques for describing individual aspects of system behaviour. But complex systems exhibit various behavioural aspects. This observation has led to research into the combination and semantic integration of specification techniques.

One such combination is CSP-OZ [10,13,14,12] integrating CSP and Object-Z. Central to CSP-OZ is the notion of a class. The specification of a class C has a format as shown in Fig. 1. The *interface* I declares channel names and types to be used by the class. Additionally, there may be *local* channels L declared that are only visible inside the class. The *CSP part* uses a CSP process P to describe the desired sequencing behaviour on the interface and local channels. As in Object-Z the state space of C is given by a nameless schema *State* : *Exp*. The initial state of C is described by a schema named *Init*. For the interface and local channels *c communication schemas* named com_*c* describe the transformation of the state space induced by communicating along c. In Z and hence Object-Z the prime decoration is used for denoting the new values after the transformation. This value may depend on values of input parameters (decorated by ?) and yield values of output parameters (decorated by !). In addition CSP-OZ uses *simple* parameters (no decoration) which are a mixture of input and output parameters (see below).



where the *OZ part Z* comprises the following schemas:

Fig. 1. Format of a CSP-OZ class.

Example 1. To illustrate the use of CSP-OZ we present here part of the specification of a *till* [45] for the problem "cash point service" defined in [6]. Informally, the till is used by inserting a card and typing in a PIN which is compared with the PIN stored on the card. In the case of a successful identification, the customer may withdraw money from the account. Upon withdrawal the bank of the customer is informed about the transaction. The till is only one component of a larger system including banks, cards, customers, and money dispensers. We specify part of the till and the bank.

Global definitions. The CSP-OZ class *Till* uses two basic Z types:

$$[Pin, CardID]$$

Pin represents the set of personal identification numbers and *CardID* the set of card identification numbers. Furthermore, we stipulate one basic type for every class which can take *reference names* to instances of this class. Reference names serve as addresses for instances of a class. They are not references in the classical sense (i.e. pointers), but merely names that uniquely identify instances. Here we need the following types

$$[TillRef, BankRef]$$

for the two classes *Till* and *Bank*.

Interface. The *Till* is connected to its environment by several typed channels (some of which are given below):

$$\begin{array}{l}
\texttt{chan}\ insertCard : [\ b? : BankRef;\ c? : CardID;\ t : \{self\}\ ] \\
\texttt{chan}\ getCustPin : [\ p? : Pin;\ t : \{self\}\ ] \\
\texttt{chan}\ getCardPin : [\ p? : Pin;\ t : \{self\}\ ] \\
\texttt{chan}\ pay : [\ a! : \mathbb{N};\ t : \{self\}\ ] \\
\texttt{chan}\ updateBalance : [\ b : BankRef;\ t : \{self\};\ a! : \mathbb{N};\ c! : CardID\ ] \\
\texttt{chan}\ stop
\end{array}$$

Every channel has a type which is given by a schema containing the (input, output or simple) parameters of the channel. Simple parameters are solely used for addressing purposes; their value may be restricted by both objects taking part in the communication over that channel. For instance, the simple parameter *t* of channel *insertCard* has as type the singleton set $\{self\}$. Upon object creation the special variable *self* is set to the reference name of the particular till. Thus, within class *C* the type of *self* is *CRef*. During the lifetime of an instance the value of *self* never changes.

CSP-OZ has also a special type *Signal* consisting only of a single value. By convention, the type Signal is omitted in a channel declaration, and semantically a communication along such a channel is identified with a pure synchronisation event, denoted by the channel name itself. The channel *stop* is an example.

In this example, the channel *insertCard* is used to insert some card with identity *c*? of a bank *b*? into the till *t*. The channel *getCustPin* expects a value *p*? of type *Pin* as its input, the channel *getCardPin* also inputs a value *p*? of type *Pin*, the channel *pay* outputs a natural number *a*! (the amount of money to be paid out to the customer), and the channel *updateBalance* is intended for communication with the bank *b* from till *t* to transmit an amount *a*! of money to be withdrawn from the account belonging to card with identity *c*!. Since there may be several banks around, the first parameter of *updateBalance* is used to identify one particular bank. The second parameter tells the bank which till was used in the

transaction (the one issuing the communication over *updateBalance*). The full definition of the till comprises more interface channels (see [45]).

CSP part. This part specifies the order in which communications along the interface channels can occur. To this end, a set of recursive process equations is given. The main equation starts with the process identifier `main`. The symbol $\stackrel{c}{=}$ is used instead of an ordinary equals symbol to distinguish between CSP process equations and Z equations.

$$
\begin{aligned}
\texttt{main} &\stackrel{c}{=} \mathit{insertCard} \rightarrow \mathit{Ident} \\
\mathit{Ident} &\stackrel{c}{=} \mathit{getCustPin} \rightarrow \mathit{getCardPin} \rightarrow \\
&\qquad (\mathit{idFail} \rightarrow \mathit{Eject} \\
&\qquad \Box\ \mathit{idSucc} \rightarrow \mathit{Service}) \\
\mathit{Eject} &\stackrel{c}{=} \mathit{eject} \rightarrow \texttt{main} \\
\mathit{Service} &\stackrel{c}{=} (\mathit{stop} \rightarrow \mathit{Eject} \\
&\qquad \Box\ \mathit{withdraw} \rightarrow \mathit{getAmount} \rightarrow \mathit{pay} \rightarrow \\
&\qquad\quad \mathit{updateBalance} \rightarrow \mathit{Eject})
\end{aligned}
$$

This process specifies the following behaviour. First the till gets the *CardID* via a communication along channel *insertCard*. Then the till checks the customer's identity by getting the customer's PIN via *getCustPin*, retrieving the PIN stored on the card via *getCardPin*, and comparing both. The communications *idFail* signals that this comparison failed, the communication *idSucc* signals that it was successful. In the case of failure the card is ejected. In the case of success that service mode is entered where the customer has the choice of stopping the interaction or withdrawing money. In the latter case the communication *getAmount* inputs the amount of money the customer wishes to withdraw, the communication *pay* initiates the money dispenser to pay out this amount, and the communication *updateBalance* informs the bank about the change.

Note that in this particular CSP process no communication values are specified. They will be dealt with in the OZ part.

OZ part. This part specifies the state of the till and the effect of the communications on this state. The state consists of the following typed variables:

[state space]
$\mathit{currCard} : \mathit{CardID}$
$\mathit{currBank} : \mathit{BankRef}$
$\mathit{currPin}, \mathit{typedPin} : \mathit{Pin}$
$\mathit{amount} : \mathbb{N}$

The effect of communications along the interface channels are specified using communication schemas. Here we give only some examples. The schema

com_*insertCard*

$\Delta(currCard, currBank)$
$b? : BankRef$
$c? : CardId$
$t : TillRef$

$currCard' = c? \wedge currBank' = b?$

specifies that a communication along channel *insertCard* may only change the state variables *currCard* and *currBank*. This is the meaning of the Δ-list in the first line of the declaration part of this schema. The second line specifies that a communication along *insertCard* has input values $c?$ of type *CardID* and $b?$ of type *BankRef*. The predicate of the schema specifies that the effects of the schema are assignments of the input values to variable *currCard* and *currBank*.

The schema

com_*idSucc*

$currPin = typedPin$

specifies that a communication *idSucc* does not change any variable of the state (no Δ-list) and is enabled only if the current PIN of the card is identical to the PIN typed in by the customer. The schema

com_*updateBalance*

$b : BankRef$
$t : TillRef$
$a! : \mathbb{N}$
$c! : CardID$

$b = currBank$
$a! = amount \wedge c! = CardId$

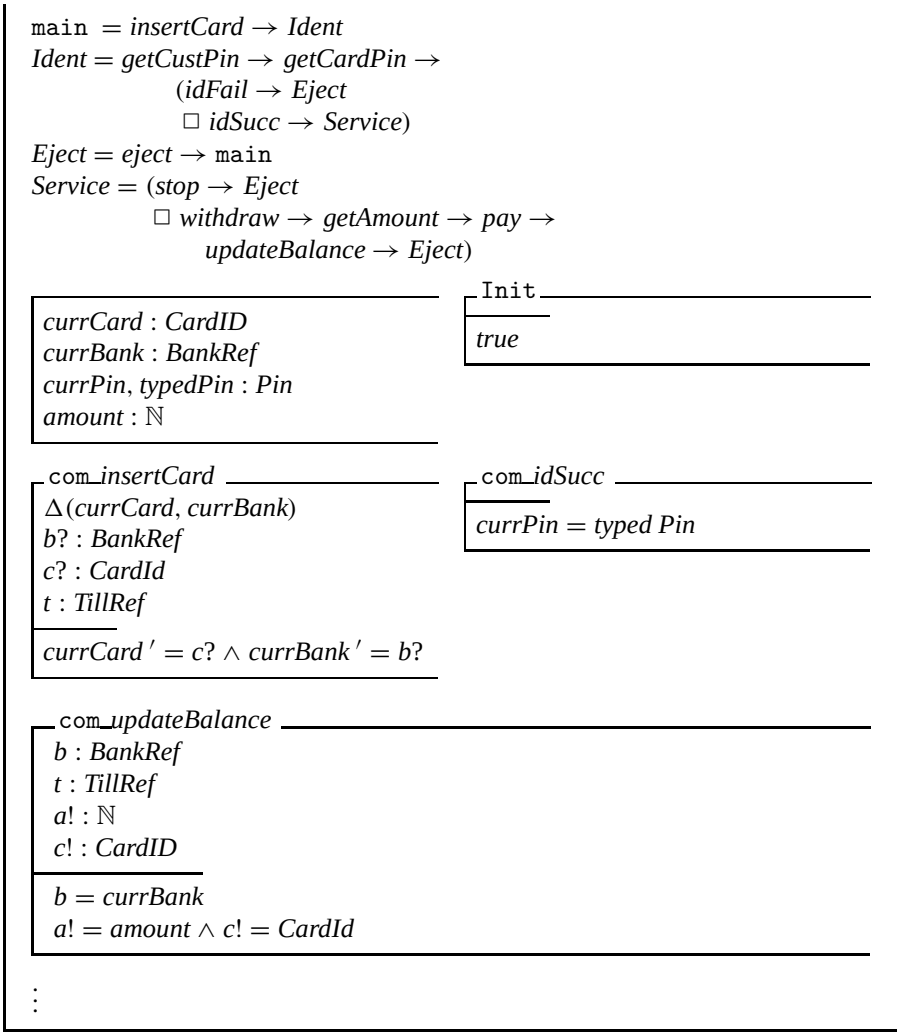
specifies that the amount $a!$ of money to be withdrawn and the identity $c!$ of the card are sent to the bank *currBank* where the balance of the account is updated. The value of the simple parameter t is — by its type declaration in the interface — restricted to the reference name *self* of the particular till instance executing *updateBalance*.

Altogether we get the following (part of the) class specification *Till*:

Till

chan *insertCard* : [$b? : BankRef$; $c? : CardID$; $t : \{self\}$]
chan *getCustPin* : [$p? : Pin$; $t : \{self\}$]
chan *getCardPin* : [$p? : Pin$; $t : \{self\}$]
chan *pay* : [$a! : \mathbb{N}$; $t : \{self\}$]
chan *updateBalance* : [$b : BankRef$; $t : \{self\}$; $a! : \mathbb{N}$; $c! : CardID$]

$\vdots$



Instances. An *instance* (or *object*) t of the class *Till* is specified by a declaration

$$t : \mathit{Till}.$$

The instance t behaves like the class *Till* with variable *self* set to t. A customer using till t might perform the following interaction with it expressed as a CSP process:

$$\mathit{Customer} \stackrel{c}{=} \mathit{insertCard}.\mathit{LZO}.765.t \to \mathit{getCustPin}.4711 \to \mathit{withdraw}.t \to \mathit{getAmount}.100 \to \mathit{SKIP}.$$

To model the behaviour of several instances $t_1, \ldots, t_n$ of the class *Till* the interleaving operator ||| of CSP can be used:

$t_1 \mid\mid\mid \cdots \mid\mid\mid t_n \quad$ or $\quad \mid\mid\mid_{i=1,\ldots,n} t_i$

These tills will be connected with a finite set of banks of the following class:

Bank

$\texttt{chan}\ updateBalance : [\ b : \{self\};\ t? : TillRef, a? : \mathbb{N};\ c? : CardID\ ]$

$\vdots$

$accounts : CardID \to \mathbb{Z}$
$transactions : CardID \to \text{seq}\ TillRef$

$\texttt{com\_}updateBalance$

$\Delta(accounts, transactions)$
$b : BankRef$
$t? : TillRef$
$a? : \mathbb{N}$
$c? : CardID$

$accounts' = accounts \oplus \{c? \mapsto accounts(c?) - a?\}$
$transactions' = transactions \oplus \{c? \mapsto transactions(c?) \frown \langle t? \rangle\}$

$\vdots$

The data domain for the first parameter of *updateBalance* is the set $\{self\}$ thus achieving correct addressing. An instance of class *Bank* can only communicate with a till via *updateBalance* when the first parameter is set to its instance name. When *updateBalance* is called the account belonging to the particular card identity $c?$ is updated and the transaction is stored.

Finally, a system comprising banks $b_1, \ldots, b_m$ connected to tills $t_1, \ldots, t_n$ can be specified by the following class:

System

$\texttt{chan}\ insertCard : [\ b? : BankRef;\ c? : CardID;\ t : TillRef\ ]$
$\texttt{chan}\ getCustPin : [\ p? : Pin;\ t : TillRef\ ]$
$\texttt{chan}\ getCardPin : [\ p? : Pin;\ t : TillRef\ ]$
$\texttt{chan}\ pay : [\ a! : \mathbb{N};\ t : TillRef\ ]$
$\texttt{local\_chan}\ updateBalance : [\ b : BankRef;\ t : TillRef;\ a! : \mathbb{N};\ c! : CardID\ ]$

$\vdots$

$\texttt{main} = (b_1, \ldots, b_m : Bank;\ t_1, \ldots, t_n : Till \bullet$
$(\mid\mid\mid_{i=1,\ldots,m} b_i) \mid\mid_{\{updateBalance\}} (\mid\mid\mid_{i=1,\ldots,n} t_i))$

where $\mid\mid_{\{updateBalance\}}$ is the parallel composition enforcing synchronisation on the channel *updateBalance* between banks and tills. The reference names in the parameters of this

channel ensure correct addressing, i.e. a bank b_j communicates with a till t_l by executing $updateBalance.b_j.t_l.v\_a.v\_c$ (with some values v_a and v_c for amount and card identity).

The class has no Object-Z part. It just contains the CSP description of the *architecture* of the system. All instances are declared local to the class and all channels which are used for internal communication between components of the system are explicitly declared as local channels.

3. Semantics

Each class of a CSP-OZ specification denotes a process obtained by transforming the OZ part into a process that runs in parallel with the CSP part. First we briefly review the semantics of CSP and Object-Z.

3.1. Semantics of CSP

The standard semantics of CSP is the $\mathcal{FD}$-semantics based on failures and divergences [34]. Starting from a set Σ of events or communications, a *failure* is a pair (s, X) consisting of a finite sequence or *trace* $s \in \Sigma^*$ and a so-called *refusal set* $X \in \mathbb{P}\,\Sigma$. Intuitively, a failure (s, X) describes that after engaging in the trace s the process can refuse to engage in any of the communications in X. Refusal sets allow us to make fine distinctions between different nondeterministic process behaviour; they are essential for obtaining a compositional definition of parallel composition in the CSP setting of synchronous communication when we want to observe deadlocks. Formally, we need the following sets of observations about process behaviour:

$$\begin{aligned} &Traces = \Sigma^*,\\ &Refusals = \mathbb{P}\,\Sigma,\\ &Failures = Traces \times Refusals. \end{aligned}$$

A *divergence* is a trace after which the process can engage in an infinite sequence of internal actions. The simplest model of CSP is the *trace semantics* $\mathcal{T}$. Let *Processes* denote the set of CSP processes. Then

$$\mathcal{T} : Processes \to \mathbb{P}\,Traces.$$

Thus $\mathcal{T}$ assigns to each CSP process a set of traces. This semantics induces a notion of *process refinement* denoted by $\sqsubseteq_{\mathcal{T}}$. For $P, Q \in Processes$ this relation is defined as follows:

$$P \sqsubseteq_{\mathcal{T}} Q \text{ iff } \mathcal{T}(P) \supseteq \mathcal{T}(Q).$$

Thus Q refines P in the trace model if Q exhibits no more traces than P. Then Q can be regarded as a process satisfying the *safety property* defined by the trace set of P.

Trace refinement is insensitive against deadlocks or divergences. To deal with these phenomena, the more sophisticated failure divergence semantics $\mathcal{FD}$ of CSP is needed. It is given by two mappings

$$\mathcal{F} : Processes \to \mathbb{P}\,Failures \text{ and } \mathcal{D} : Processes \to \mathbb{P}\,Traces.$$

For a CSP process P we define $\mathcal{FD}(P) = (\mathcal{F}(P), \mathcal{D}(P))$. Certain well-formedness conditions relate the values of $\mathcal{F}$ and $\mathcal{D}$ (see [34], p. 192). Also the $\mathcal{FD}$-semantics induces a notion of process refinement denoted by $\sqsubseteq_{\mathcal{FD}}$. For $P, Q \in \mathit{Processes}$ this relation is defined as follows:

$$P \sqsubseteq_{\mathcal{FD}} Q \text{ iff } \mathcal{F}(P) \supseteq \mathcal{F}(Q) \text{ and } \mathcal{D}(P) \supseteq \mathcal{D}(Q).$$

Thus Q refines P in the failure divergence model if Q is more deterministic (i.e. has fewer failures) and is more defined (i.e. less divergent) than P. In particular, if P represents a specification without divergences then Q refining P is guaranteed to exhibit "liveness", i.e. to react to communications as required by P without getting lost in infinite internal process activity.

3.2. Semantics of communication schemas

For Object-Z a history semantics based on sequences of states and events (operation calls) as well as some more abstract semantics are defined [37]. We do not need these semantics here because we only use the state transformation view of the communication schemas in the OZ part. A communication schema

com_c

$\Delta(\overline{x})$
$in? : D_{in}$
$s : D_s$
$out! : D_{out}$

$p(\overline{x}, in?, s, out!, \overline{x}')$

with Δ-list $\overline{x}$, input parameters $in?$, simple parameters s, and output parameters $out!$ describes a transformation on the state space as specified by the predicate $p(\overline{x}, in?, s, out!, \overline{x}')$ where additionally $y' = y$ holds for all state variables y not mentioned in the Δ-list. The transformation is defined only for those values of $\overline{x}$, s and $in?$ satisfying the *precondition* of com_c requiring that there exist corresponding values of $out!$ and $\overline{x}'$. If the transformation is defined any of these values of $out!$ and $\overline{x}'$ can be chosen nondetermistically.

Z comes with the usual notion of *data refinement* and operation refinement [48,7]. Given a relation ρ between an abstract and a concrete state space, a concrete communication schema C_com_c refines an abstract communication schema A_com_c, denoted by

$$A\_com\_c \sqsubseteq_{\rho} C\_com\_c,$$

if C_com_c is more defined and more deterministic than A_com_c.

3.3. Semantics of CSP-OZ

The semantics of CSP-OZ is defined in [10,12]. Each CSP-OZ class denotes a process in the failure divergence model of CSP. For this purpose, method invocations are identified with events in the CSP sense. The process is obtained by transforming the OZ part of a

class into a CSP process that runs in parallel and communicates with the CSP part of the class.

Consider a CSP-OZ class C as in Fig. 1. Let st denote the list of variables declared in the schema *State*, and let st' be the corresponding list of variables of the decorated schema *State*$'$. For a channel c declared in I or L let in_c, $simple_c$, out_c denote the list of input, simple, and output parameters of this channel. Note that the communication schema $\mathtt{com\_}c$ depends on the values of st, in_c, $simple_c$, out_c, st'. For a given list ℓ of typed variables let $Val(\ell)$ denote the set of all corresponding lists of values that these variables may assume according to their type.

Transformation. The OZ part of the class is transformed into a CSP process *OZMain* defined by the following system of (parameterised) recursive equations for *OZPart* using the (indexed) CSP operators for internal nondeterministic choice ($\sqcap$) and alternative composition ($\Box$):

$$\begin{aligned} OZMain = {} & \textstyle\prod_{v\_st} OZPart(v\_st) \\ OZPart(v\_st) = {} & \textstyle\Box_{c,\ v\_in_c,\ v\_simple_c} \\ & \textstyle\prod_{v\_out_c,\ v\_st'} c.v\_in_c.v\_simple_c.v\_out_c \rightarrow OZPart(v\_st') \end{aligned}$$

where in the first equation the index v_st of the $\sqcap$ operator ranges over all value lists in $Val(st)$ that satisfy *Init*. Thus the process *OZMain* can nondeterministically choose any values for the state variables st that satisfy *Init* to start with. For the $\Box$ operator in the second equation, the index c ranges over all channels declared in I and L, the index v_in_c ranges over all value lists in $Val(in_c)$, and the index v_simple_c ranges over all value lists in $Val(simple_c)$ such that the precondition of the communication schema for c, i.e.

$$\exists\, out_c;\ st' \bullet \mathtt{com\_}c,$$

holds for these values. Finally, for any chosen c and values v_in_c, v_simple_c, the indices of the subsequent $\sqcap$ operator are determined as follows: the index v_out_c ranges over all value lists in $Val(out_c)$ and the index v_st' ranges over all value lists in $Val(st')$ such that

$$\mathtt{com\_}c$$

holds for these values. So the $OZPart(v\_st)$ is ready for every communication event $c.v\_in_c.v\_simple_c.v\_out_c$ along a channel c in I or L where for the values v_in_c, v_simple_c the communication schema $\mathtt{com\_}c$ is satisfiable for some output values v_out_c and successor state v_st'. For given input values v_in_c, v_simple_c any such v_out_c and v_st' can be nondeterministically chosen to yield $c.v\_in_c.v\_simple_c.v\_out_c$ and the next recursive call $OZPart(v\_st')$. Thus input and output along channels c are modelled by a subtle interplay of the CSP alternative and nondeterministic choice.

Semantics of a class. Using parallel composition and hiding the semantics of a class is defined in the failure divergence model as follows:

$$\mathcal{FD}(C) = \mathcal{FD}((P \ ||_{\{commonEvents\}} \ OZMain) \setminus Events(L)).$$

Here $||_A$ is the parallel composition with synchronisation on the event set A and $\backslash B$ is the hiding operator concealing all events in the event set B (see e.g. [34]). In this semantic equation we refer to the synchronisation set

$$commonEvents = \alpha(P) \cap \alpha(OZMain)$$

where $\alpha(Q)$ is the alphabet of a given CSP process Q, i.e. the set of all communications in which Q can engage. $Events(L)$ is the set of all communications that are possible on the local channels L.

Semantics of self. So far, this semantic definition treats classes without occurrence of the variable *self*. The name *self* is used as a reference name for a particular instance. However, it is simply a name and not a reference in the sense of a pointer. In contrast to Object-Z, CSP-OZ adopts no reference semantics for instances but a value semantics. Instances of classes may be declared at two points within a specification. The first possibility is a variable in the state space of a class C_1 which has the type of another class C_2. Then C_2 must be a pure Object-Z class, i.e. without a CSP part, and the semantics is the value semantics also used in earlier work on Object-Z. We will therefore not explicitly discuss it here. The second possibility is to instantiate classes within the CSP part (and then there are no restrictions on the classes). Since CSP-OZ objects communicate via channels (and not via mutual references to each other) there is, however, still no necessity of defining a reference semantics. We only have to ensure that there is a way to communicate with a particular object, and for this purpose the variable *self* is introduced. The existence of *self* allows us to uniquely address an instance. To this end the address can become part of a communication event (i.e. can be one parameter of a channel).

Since *self* is different for every instance of a class the events of one instance differ from the events of other instances (of the same class). This has to be reflected in our semantics. When *self* is used the above transformation of Object-Z to CSP therefore has to be modified slightly. Instead of a CSP process *OZMain*, a *parameterised* process $OZMain(v\_self)$ is now the result of the transformation where *v_self* ranges over the values of the variable *self*, i.e. the set *CRef* of reference names of class C.

$$\begin{aligned} OZMain(v\_self) = {} & \prod_{v\_st} OZPart(v\_st, v\_self) \\ OZPart(v\_st, v\_self) = {} & \Box_{c,\ v\_in_c, v\_simple_c} \\ & \prod_{v\_out_c,\ v\_st'} c.v\_in_c.v\_simple_c.v\_out_c \rightarrow \\ & \qquad OZPart(v\_st', v\_self) \end{aligned}$$

where the predicate $\mathtt{com}\_c$ depends on the values of $st, in_c, simple_c, out_c, st'$ and *self* may be one of the simple parameters in the list $simple_c$. For instance, for the channel *updateBalance* the (interior of the) translation yields

$$updateBalance.v\_self.v\_t.v\_a.v\_c \rightarrow OZPart(v\_st', v\_self).$$

The semantics of a class C is thus a function $\mathcal{FD}(C) : CRef \rightarrow \mathbb{P}\, Failures \times \mathbb{P}\, Traces$ from reference names (of class C) to failure divergence sets.

Semantics of an instance. Suppose an instance o of a CSP-OZ class C is declared by $o : C$ (within a CSP process). Then o denotes a process which is obtained by calling $\mathcal{FD}(C)$ with the reference name o in place of the formal parameter *self*:

$$\mathcal{FD}(o) = \mathcal{FD}(C)(o).$$

Alternatively, the notation $C(o)$ can be used for instantiation. Then by definition

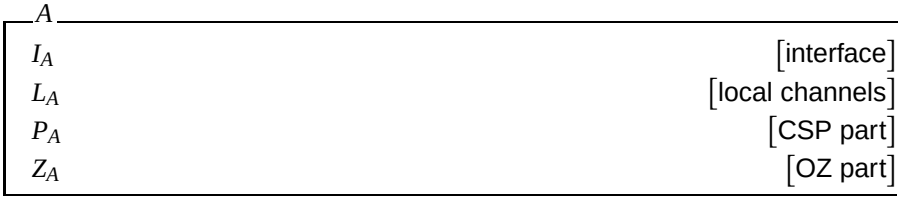
$$\mathcal{FD}(C(o)) = \mathcal{FD}(C)(o).$$

Refinement compositionality. By the above process semantics of CSP-OZ, the refinement notion $\sqsubseteq_{\mathcal{FD}}$ is immediately available for CSP-OZ. In [12] it has been shown that CSP-OZ satisfies the principle of *refinement compositionality*, i.e. refinement of the parts implies refinement of the whole. Formally:

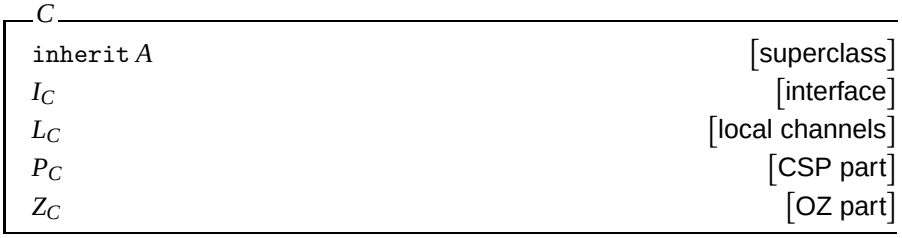
- Process refinement $P_1 \sqsubseteq_{\mathcal{FD}} P_2$ implies refinement in CSP-OZ:
  $\texttt{spec}\ I\ L\ P_1\ Z\ \texttt{end} \quad \sqsubseteq_{\mathcal{FD}} \quad \texttt{spec}\ I\ L\ P_2\ Z\ \texttt{end}$
- Data refinement $Z_1 \sqsubseteq_{\rho} Z_2$ for a refinement relation ρ (as verified by downward simulation conditions) implies refinement in CSP-OZ:
  $\texttt{spec}\ I\ L\ P\ Z_1\ \texttt{end} \quad \sqsubseteq_{\mathcal{FD}} \quad \texttt{spec}\ I\ L\ P\ Z_2\ \texttt{end}$

4. Inheritance

Process refinement $P \sqsubseteq_{\mathcal{FD}} Q$ in CSP stipulates that P and Q have the *same* interface. Often one wishes to *extend* the communication capabilities of a process or the operation capabilities of a class. This can be specified using the notion of *inheritance*, a syntactic relationship on classes: a superclass (or abstract class) A is extended to a subclass (or concrete class) C. The subclass should inherit the parts of the superclass. In CSP-OZ this is denoted as follows. Given a superclass A of the form



we obtain a subclass C of A by referring to A using the `inherit` clause:



The semantics of the `inherit` operator is defined in a transformational way, i.e. by incorporating the superclass A into C yielding the following expanded version of C:

C

I	[interface]
L	[local channels]
P	[CSP part]
Z	[OZ part]

where $I = I_A \cup I_C$, $L = L_A \cup L_C$ and P is obtained from P_A and P_C by parallel composition and Z is obtained from Z_A and Z_C by schema conjunction.

More precisely, to obtain the CSP part P we first replace in P_A and P_C the process identifiers `main` by new identifiers $\texttt{main}_A$ and $\texttt{main}_C$ respectively, then collect the resulting set of process equations, and add the equation

$$\texttt{main} \stackrel{C}{=} \texttt{main}_A \;||_{\{commonEvents\}}\; \texttt{main}_C$$

modelling parallel composition of P_A and P_C with synchronisation on their common events. To obtain the OZ part Z the corresponding schemas of Z_A and Z_C are conjoined:

$$\begin{array}{l} State = State_A \wedge State_C, \\ Init = Init_A \wedge Init_C, \\ \texttt{com\_}c = \texttt{com\_}c_A \;\wedge\; \texttt{com\_}c_C \quad \text{for all channels } c \text{ in } (I_A \cup L_A) \cap (I_C \cup L_C), \\ \texttt{com\_}c = \texttt{com\_}c_A \quad \text{for all channels } c \text{ in } (I_A \cup L_A) \setminus (I_C \cup L_C), \\ \texttt{com\_}c = \texttt{com\_}c_C \quad \text{for all channels } c \text{ in } (I_C \cup L_C) \setminus (I_A \cup L_A). \end{array}$$

Example 2. We wish to extend the specification of the basic till of Example 1 with the option to switch the language of the display (English vs. German) at any time. To this end, we introduce the type

[*Language*]

and require that it has at least two different values:

$$g, e : Language \;\vdash\; g \neq e$$

The following class *ExtTill* inherits the specification of the class *Till*:

ExtTill

`inherit` *Till*

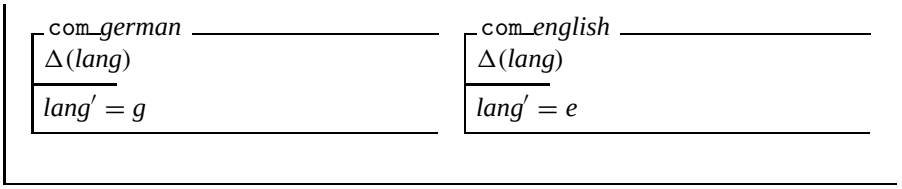
`chan` *english*, *german*

$\texttt{main} = german \rightarrow english \rightarrow \texttt{main}$

$lang : Language$

`Init`

$lang = e$



Semantically, the CSP part of *ExtTill* runs in parallel with the inherited CSP part of *Till* thus allowing us to switch the language at any time.

5. Inheritance of properties

In this section we study the preservation of properties under inheritance. The scenario we are interested in is the following: suppose we have a superclass A for which we have already verified that a certain property P holds. Now we extend A to a subclass C and would like to know under which conditions P also holds for C. This would allow us to check the conditions on the subclass and avoid re-verification.

Since CSP-OZ has a failure divergence semantics, we use the CSP style of property specification. In CSP, properties are formalised by CSP processes. A property P holds for a class A if the class refines the property. We are thus interested in a reasoning similar to that for refinement. In CSP, properties are preserved under refinement, i.e. if P holds for A and A is refined by C then P holds for C. This is due to the transitivity of the refinement relation $\sqsubseteq_{\mathcal{FD}}$:

$$P \sqsubseteq_{\mathcal{FD}} A \wedge A \sqsubseteq_{\mathcal{FD}} C \Rightarrow P \sqsubseteq_{\mathcal{FD}} C.$$

If inheritance is employed (C being a subclass of A) instead of refinement this is in general not true any more because inheritance allows us to modify essential aspects of a class and thus may destroy properties proven for the superclass. We thus have to require a closer relationship between super- and subclass in order to achieve inheritance of properties. A relationship which guarantees a certain form of property inheritance is *behavioural subtyping* [27]. Originally studied in state-based contexts, this concept has recently been extended to behaviour-oriented formalisms (see [15]) and is thus adequate for CSP-OZ with its failure divergence semantics. Behavioural subtyping guarantees substitutability while also allowing extension of functionality as introduced by inheritance.

In the following we will look at two forms of behavioural subtyping, defined in the two semantic models of CSP, trace and failure divergence model. We show that the trace-based notion of subtyping preserves safety properties and the failure-divergence-based notion preserves a form of liveness properties. Originally, these two forms of behaviour-oriented subtyping have been defined as relations between failure and divergence sets [15,47]. For ease of understanding, we will use an alternative definition here which relies on the application of CSP operators on the (semantics of) classes. These alternative definitions have been developed in order to be able to apply the FDR model-checker for checking subtype relationships [46]. In Section 5.3 we will also briefly discuss the relationship

between inheritance and subtyping, and describe a simple pattern for the Object-Z part which guarantees that inheritance leads to subtypes. All proofs can be found in Section 6.

5.1. Safety: trace properties

Since CSP offers different forms of refinement there are also different forms of satisfaction. We say that a class satisfies a property with respect to safety issues if it is a *trace refinement* of the property; when failure divergence refinement is used a (limited form of) liveness is checked (see next section).

Definition 1. Let A be a class and P a CSP property (process). *A satisfies the trace property P* (A satisfies P in the trace model) iff

$$\mathcal{T}(A) \subseteq \mathcal{T}(P)$$

(or equivalently $P \sqsubseteq_{\mathcal{T}} A$) holds.

We illustrate this by means of the cash point example. Consider the following class A_0 with a behaviour as specified in Fig. 2. For reasons of readability we consider only a very simple form of till.

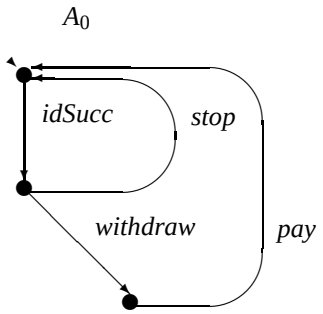


Fig. 2. The simple till A_0.

We want to specify that money is paid out only after the correct PIN code has been entered. As a CSP property process this is:

$$Seq = idSucc \rightarrow pay \rightarrow Seq$$
$$Safe = Seq \;|||\; CHAOS(\,\Sigma - \{idSucc, pay\}\,)$$

Here the process $CHAOS(\alpha)$, where α is a set of events, is the chaotic process which can always choose to communicate as well as refuse events of α [34]. $CHAOS(\alpha)$ is defined by the recursive equation

$$CHAOS(\alpha) = STOP \sqcap (\ \Box\, ev \in \alpha \bullet ev \rightarrow CHAOS(\alpha)). \tag{1}$$

A_0 satisfies the trace property *Safe* since $Safe \sqsubseteq_{\mathcal{T}} A_0$.

Next we would like to know whether such a trace property P can be inherited to a subclass (or more specifically, a subtype) C. As a first observation we notice that C potentially has traces over a larger alphabet than A since it can have a larger interface. This might immediately destroy the holding of a trace property. Nevertheless, a trace property might still hold in the sense that, as far as the operations of A are concerned, the ordering of operations as specified in P also holds in C. This can be tested by *projecting* (the traces or failures and divergences of) both the class C and the property P down to the alphabet of A.

We thus define the following projection operator for traces s and sets α of events. Let the *projection* $s \downarrow \alpha$ be the trace that results from s by removing all elements outside the set α. Obviously, projection distributes over concatenation, i.e. for traces s, t we have

$$(s \frown t) \downarrow \alpha = (s \downarrow \alpha) \frown (t \downarrow \alpha).$$

We lift this operation to CSP processes P in the trace and failure divergence model by applying it elementwise to traces, failures and divergences:

$$\begin{array}{lcl}
\mathcal{T}(P \downarrow \alpha) & = & \{s : \mathit{Traces} \mid s \in \mathcal{T}(P) \bullet s \downarrow \alpha)\} \\
\mathcal{F}(P \downarrow \alpha) & = & \{s : \mathit{Traces};\ X, Z : \mathit{Refusals} \mid Z \subseteq \Sigma - \alpha \wedge \\
 & & \quad (s, X) \in \mathcal{F}(P)\ \bullet (s \downarrow \alpha, X \cup Z)\} \cup \\
 & & \{s : \mathit{Traces};\ X : \mathit{Refusals} \mid s \in \mathcal{D}(P \downarrow \alpha) \bullet (s, X)\} \\
\mathcal{D}(P \downarrow \alpha) & = & \{s, v : \mathit{Traces} \mid s \in \mathcal{D}(P) \bullet (s \downarrow \alpha) \frown v)\}.
\end{array}$$

Note that here and elsewhere we use a Z style notation for sets (see e.g. [48]). The refusals $X \cup Z$ with $Z \subseteq \Sigma - \alpha$ reflect the idea that outside the projection alphabet α any event can be refused. The failures and divergence sets are constructed in such a way that the closure properties of these models are again fulfilled: divergent traces can always be extended, and after a divergence anything can be refused. For a more detailed account of these issues see [34].

This operator is now used to formulate satisfaction of a property with respect to an alphabet.

Definition 2. Let C be a class, P a property and $\alpha \subseteq \Sigma$ a set of events. C satisfies the *trace property P with respect to α* iff

$$\mathcal{T}(C \downarrow \alpha) \subseteq \mathcal{T}(P \downarrow \alpha)$$

or equivalently $P \downarrow \alpha \sqsubseteq_{\mathcal{T}} C \downarrow \alpha$ holds.

In the sequel we use the following notation. Let A and C be classes with $\alpha(A) \subseteq \alpha(C)$ and

$$N = \alpha(C) - \alpha(A)$$

denote the set of new methods. The question we would ultimately like to answer is thus as follows: if A satisfies P in the trace model, does C satisfy P in the trace model w.r.t. $\alpha(A)$? This is in fact the case when C is a *trace subtype* of A.

Definition 3. C is a *trace subtype* of A, abbreviated $A \sqsubseteq_{tr-st} C$, iff

$$A \,|||\, CHAOS(N) \sqsubseteq_{\mathcal{T}} C.$$

Intuitively, the interleaving parallel composition of A with the chaotic process over the set N says that C may have new methods N in addition to A and that these can at any time be executed as well as refused, but they have to be independent from (interleaved with) the A-part. Note that trace subtyping corresponds to trace refinement if $N = \varnothing$, i.e. if C has no additional methods.[1]

Safety properties are inherited by trace subtypes as the following theorem shows.

Theorem 1. *Let A, C be processes with $A \sqsubseteq_{tr-st} C$, and let P be a process formalising a trace property. If A satisfies P in the trace model then C satisfies P in the trace model w.r.t. $\alpha(A)$.*

As an example we look at an extension of class A_0. The till C_0 depicted in Fig. 3 extends A_0 with a facility of viewing the current balance of the account. C_0 is a trace subtype of A_0 and thus inherits the property *Safe*, i.e. C_0 satisfies *Safe* w.r.t. $\alpha(A_0)$.

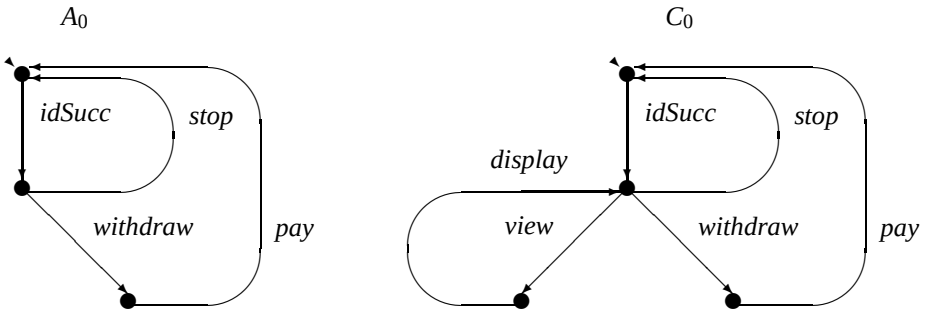


Fig. 3. A class and a trace subtype.

It can be shown that with respect to the trace refinement ordering $\sqsubseteq_{\mathcal{T}}$ the process $A \,|||\, CHAOS(N)$ is the *smallest* process inheriting all properties of A (with respect to $\alpha(A)$).

Theorem 2. *$C = A \,|||\, CHAOS(N)$ is w.r.t. $\sqsubseteq_{\mathcal{T}}$ the smallest process such that for all processes P*

$$P \sqsubseteq_{\mathcal{T}} A \quad \textit{implies} \quad P \downarrow \alpha(A) \sqsubseteq_{\mathcal{T}} C \downarrow \alpha(A).$$

[1] This is because $CHAOS(\varnothing)$ is equivalent to *STOP*, and *STOP* is the neutral element for interleaving provided the termination event ✓ is absent.

5.2. "Liveness": $\mathcal{FD}$ properties

Liveness properties are checked in CSP by comparing the property process and the class with respect to their failure divergence set, i.e. checking whether the class is a failure divergence refinement of the property. This yields a form of *bounded* liveness check: it can be proven that methods can be refused or, conversely, are always enabled after certain traces, without the danger of the class getting lost in an infinite internal activity. Unbounded liveness, e.g. eventuality properties on traces expressible in temporal logic, cannot be specified in CSP.

Definition 4. Let A be a class and P a CSP property (process). *A satisfies the liveness property P* (or A satisfies P in the $\mathcal{FD}$ model) iff

$$\mathcal{FD}(A) \subseteq \mathcal{FD}(P)$$

or equivalently $P \sqsubseteq_{\mathcal{FD}} A$ holds.

We illustrate this again with our till example. The property we would like to prove for class A_0 concerns service availability: after the PIN code has been verified the withdrawal of money cannot be refused. Formalised as a CSP property this is:

$$\begin{aligned} \mathit{Live} = \mathit{idSucc} \to \mathit{withdraw} \to \mathit{Live} \\ \qquad \Box\ (\mathit{STOP} \\ \qquad\quad \sqcap\ \mathit{pay} \to \mathit{Live} \\ \qquad\quad \sqcap\ \mathit{stop} \to \mathit{Live}) \\ \quad \sqcap\ \mathit{pay} \to \mathit{Live} \\ \quad \sqcap\ \mathit{stop} \to \mathit{Live}. \end{aligned}$$

Class A_0 satisfies the liveness property *Live* since $\mathit{Live} \sqsubseteq_{\mathcal{FD}} A_0$ holds.

Analogously to trace properties, we define now the satisfaction of a liveness property with respect to an alphabet.

Definition 5. Let C be a class, P a property and $\alpha \subseteq \Sigma$ a set of events. C satisfies the *liveness property P with respect to alphabet α* iff

$$\mathcal{FD}(C \downarrow \alpha) \subseteq \mathcal{FD}(P \downarrow \alpha)$$

or equivalently $P \downarrow \alpha \sqsubseteq_{\mathcal{FD}} C \downarrow \alpha$ holds.

Liveness properties can be shown to be inherited to classes which are *failure divergence subtypes*[2] of the superclass.

Definition 6. C is a *failure divergence subtype* of A, abbreviated $A \sqsubseteq_{fd-st} C$, iff

$$A \,|||\, \mathit{CHAOS}(N) \sqsubseteq_{\mathcal{FD}} C.$$

[2] Failure divergence subtyping coincides with *optimal subtyping* of [15] and [45].

This definition lifts the idea of trace subtyping to the failure divergence semantics: in class C anything "new" is allowed in parallel with the behaviour of A as long as it does not interfere with the "old" part. Looking at class C_0 in comparison with A_0 we find that C_0 is not a failure divergence subtype of A_0. For instance, C_0 has the pair $(\langle idSucc, view\rangle, \{\Sigma \setminus \{display\}\})$ in its failure set for which no corresponding pair can be found in $\mathcal{F}(A_0 \mid\mid\mid CHAOS(\{view, display\}))$ (the crucial point is the refusal of *withdraw* by C_0). Moreover, C_0 does not satisfy the property *Live*.

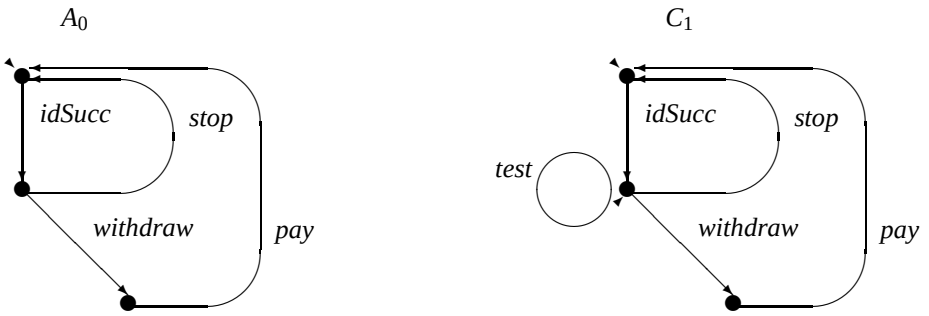


Fig. 4. A class and a failure divergence subtype.

Class C_1 as depicted in Fig. 4 on the other hand is a failure divergence subtype of class A_0 and indeed it inherits property *Live*.

Theorem 3. *Let A, C be classes with $A \sqsubseteq_{fd-st} C$, and let P be a process formalising a liveness property. If A satisfies P in the $\mathcal{F}D$ model, then C satisfies P in the $\mathcal{F}D$ model w.r.t. $\alpha(A)$.*

A more complex extension of the simple till is shown in Fig. 5. As in Example 2 the extension allows switching between different languages in the display, but switching does not affect the basic functionality. Since C_2 turns out to be a failure divergence subtype of A_0, we deduce by the previous theorem that C_2 satisfies *Live*.

Again, $A \mid\mid\mid CHAOS(N)$ is the smallest process, now in the $\mathcal{F}D$-refinement ordering, which inherits all the properties of A.

Theorem 4. *$C = A \mid\mid\mid CHAOS(N)$ is w.r.t. $\sqsubseteq_{\mathcal{F}D}$ the smallest process such that for all processes P*

$$P \sqsubseteq_{\mathcal{F}D} A \quad \textit{implies} \quad P \downarrow \alpha(A) \sqsubseteq_{\mathcal{F}D} C \downarrow \alpha(A).$$

5.3. Inheritance and subtyping

Inheritance is primarily a concept supporting the re-use of specification and code. However, if correctness is of interest the results of the last section can be used. Thus it is necessary to know when a subclass is a subtype. For the CSP part of a CSP-OZ class this can easily be checked with the FDR model-checker [46]. For the Object-Z part the same

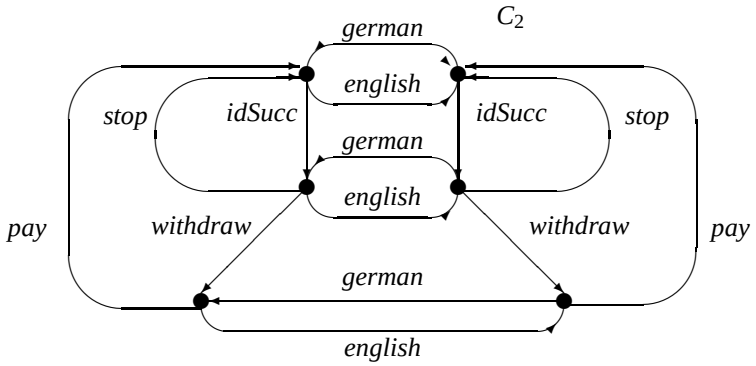


Fig. 5. Another failure divergence subtype.

check can be applied once it has been translated to CSP as defined in Section 3. In addition there are *construction patterns* which — when applied — automatically yield subtypes. In the following we present one such pattern. The pattern applies to the Object-Z part; patterns for the CSP part can be found in [45].

Subtyping can be seen as a combination of refinement and extension. Therefore downward simulation conditions for data refinement[3] can be applied when a subclass does not extend the interface of its superclass (i.e. does not specify new methods). Since downward simulation implies failure divergence refinement, these conditions are sufficient for showing that a (non-extending) subclass is a failure divergence subtype of its superclass. If there are new methods in the subtype (i.e. $N \neq \varnothing$) additional conditions on these methods have to guarantee that a subtype is achieved. A very simple condition is that new methods may not modify variables already present in the superclass.

Theorem 5. *Let A, C be classes, with C a subclass of A, and let again $N = \alpha(C) - \alpha(A)$. Then C is a failure divergence subtype of A if C restricted to $\alpha(A)$ is a downward simulation of A, and for all $c \in N$ the following non-modification condition holds:*

the Δ-list of $\texttt{com\_}c$ *contains no variables of* $State_A$.

The proof of this theorem can be found in [45].

6. Proofs of the theorems

This section presents the proofs of the four theorems on property inheritance. We start with the definition of some CSP processes and operators needed in the proofs. Furthermore, we show some algebraic properties of the projection operator.

[3] Upward simulation cannot be used since this does not induce failure divergence refinement [2].

6.1. Definitions

As in Section 3 let Σ be the set of all events, $\mathit{Traces} = \Sigma^*$, and $\mathit{Refusals} = \mathbb{P}\,\Sigma$.

Basic processes. We need the CSP processes *STOP* and $\mathit{CHAOS}(\alpha)$ for a set α of events in the trace and failure divergence model:

$$\begin{aligned}
&\mathcal{T}(\mathit{STOP}) = \{\langle\rangle\}\\
&\mathcal{F}(\mathit{STOP}) = \{X : \mathit{Refusals} \bullet (\langle\rangle, X)\}\\
&\mathcal{D}(\mathit{STOP}) = \varnothing.
\end{aligned}$$

$\mathit{CHAOS}(\alpha)$ as defined by the recursive Eq. (1) in Section 5 yields the following explicit representation:

$$\begin{aligned}
&\mathcal{T}(\mathit{CHAOS}(\alpha)) = \alpha^*\\
&\mathcal{F}(\mathit{CHAOS}(\alpha)) = \{s : \mathit{Traces};\ X : \mathit{Refusals} \mid s \in \alpha^* \bullet (s, X)\}\\
&\mathcal{D}(\mathit{CHAOS}(\alpha)) = \varnothing.
\end{aligned}$$

Interleaving. Let $s \mid\mid\mid t$ denote the set of all interleavings of given traces s and t. For $s, t \in \Sigma^*$ and $a, b \in \Sigma$ this set is defined inductively as follows:

$$\begin{aligned}
\langle\rangle \mid\mid\mid t &= \{t\}\\
s \mid\mid\mid \langle\rangle &= \{s\}\\
\langle a\rangle \frown s \mid\mid\mid \langle b\rangle \frown t &= \{u : \mathit{Traces} \mid u \in s \mid\mid\mid \langle b\rangle \frown t \bullet \langle a\rangle \frown u\} \cup\\
&\quad \{u : \mathit{Traces} \mid u \in \langle a\rangle \frown s \mid\mid\mid t \bullet \langle b\rangle \frown u\}.
\end{aligned}$$

For CSP processes P, Q the interleaving operator is defined as follows:

$$\begin{aligned}
\mathcal{T}(P \mid\mid\mid Q) &= \{s, t, u : \mathit{Traces} \mid s \in \mathcal{T}(P) \wedge t \in \mathcal{T}(Q) \wedge u \in s \mid\mid\mid t \bullet u\}\\
\mathcal{F}(P \mid\mid\mid Q) &= \{s, t, u : \mathit{Traces};\ X, Y : \mathit{Refusals} \mid\\
&\qquad (s, X) \in \mathcal{F}(P) \wedge (t, Y) \in \mathcal{F}(Q) \wedge u \in s \mid\mid\mid t \bullet (u, X \cap Y)\} \cup\\
&\quad \{s : \mathit{Traces};\ X : \mathit{Refusals} \mid s \in \mathcal{D}(P \mid\mid\mid Q) \bullet (s, X)\}\\
\mathcal{D}(P \mid\mid\mid Q) &= \{s, t, u, v : \mathit{Traces} \mid (s, \varnothing) \in \mathcal{F}(P) \wedge (t, \varnothing) \in \mathcal{F}(Q) \wedge\\
&\qquad (s \in \mathcal{D}(P) \vee t \in \mathcal{D}(Q)) \wedge u \in s \mid\mid\mid t \bullet u \frown v\}.
\end{aligned}$$

Projection. The projection operator $P \downarrow \alpha$ was defined in Section 5. Since this is not a standard CSP operator, we first prove some basic algebraic properties of projection concerning distribution over interleaving.

Lemma 1. *In the trace model of CSP projection distributes over interleaving, i.e. for all CSP processes P, Q and alphabets α*

$$(P \mid\mid\mid Q) \downarrow \alpha \ =_{\mathcal{T}}\ P \downarrow \alpha \mid\mid\mid Q \downarrow \alpha.$$

Proof. We have to show

$$\mathcal{T}((P \mid\mid\mid Q) \downarrow \alpha) = \mathcal{T}(P \downarrow \alpha \mid\mid\mid Q \downarrow \alpha).$$

This equation rests on the fact that on individual traces projection distributes over interleaving:

$$\exists\, u : \mathit{Traces} \bullet u \in s \mid\mid\mid t \wedge \bar{u} = u \downarrow \alpha \Leftrightarrow \bar{u} \in (s \downarrow \alpha) \mid\mid\mid (t \downarrow \alpha). \tag{2}$$

Then we deduce the following chain of equations:

$$\begin{array}{ll} & \mathcal{T}((P \mid\mid\mid Q) \downarrow \alpha) \\ = & \quad \{ \text{ definition of } \downarrow \alpha \ \} \\ & \{u : \mathit{Traces} \mid u \in \mathcal{T}(P \mid\mid\mid Q) \bullet u \downarrow \alpha)\} \\ = & \quad \{ \text{ definition of } \mid\mid\mid \ \} \\ & \{s, t, u : \mathit{Traces} \mid s \in \mathcal{T}(P) \wedge t \in \mathcal{T}(Q) \wedge u \in s \mid\mid\mid t \bullet u \downarrow \alpha)\} \\ = & \quad \{\text{property (2)}\} \\ & \{s, t, \bar{u} : \mathit{Traces} \mid s \in \mathcal{T}(P) \wedge t \in \mathcal{T}(Q) \wedge \bar{u} \in (s \downarrow \alpha) \mid\mid\mid (t \downarrow \alpha) \bullet \bar{u}\} \\ = & \quad \{ \text{ definition of } \downarrow \alpha \ \} \\ & \{\bar{s}, \bar{t}, \bar{u} : \mathit{Traces} \mid \bar{s} \in \mathcal{T}(P \downarrow \alpha) \wedge \bar{t} \in \mathcal{T}(Q \downarrow \alpha) \wedge \bar{u} \in \bar{s} \mid\mid\mid \bar{t} \bullet \bar{u}\} \\ = & \quad \{ \text{ definition of } \mid\mid\mid \ \} \\ & \mathcal{T}(P \downarrow \alpha \mid\mid\mid Q \downarrow \alpha). \quad \square \end{array}$$

Lemma 2. *In the failure divergence model of CSP, projection sub-distributes over interleaving, i.e. for all CSP processes P, Q and alphabets α*

$$P \downarrow \alpha \mid\mid\mid Q \downarrow \alpha \sqsubseteq_{\mathcal{FD}} (P \mid\mid\mid Q) \downarrow \alpha.$$

Proof. More precisely, we show

(i) $\mathcal{F}((P \mid\mid\mid Q) \downarrow \alpha) \subseteq \mathcal{F}(P \downarrow \alpha \mid\mid\mid Q \downarrow \alpha)$
(ii) $\mathcal{D}((P \mid\mid\mid Q) \downarrow \alpha) = \mathcal{D}(P \downarrow \alpha \mid\mid\mid Q \downarrow \alpha)$.

Again we use the fact (2) that on traces projection distributes over interleaving.

Re (i): We deduce the following chain of equations:

$$\begin{array}{ll} & \mathcal{F}((P \mid\mid\mid Q) \downarrow \alpha) \\ = & \quad \{ \text{ definition of } \downarrow \alpha \ \} \\ & \{u : \mathit{Traces};\ X, Z : \mathit{Refusals} \mid Z \subseteq \Sigma - \alpha \wedge \\ & \quad (u, X) \in \mathcal{F}(P \mid\mid\mid Q) \bullet (u \downarrow \alpha, X \cup Z)\} \cup \\ & \{s : \mathit{Traces};\ X : \mathit{Refusals} \mid s \in \mathcal{D}((P \mid\mid\mid Q) \downarrow \alpha) \bullet (s, X)\} \\ = & \quad \{ \text{ definition of } \mid\mid\mid \ \} \\ & \{s, t, u : \mathit{Traces};\ X, Y, Z : \mathit{Refusals} \mid Z \subseteq \Sigma - \alpha \wedge \\ & \quad (s, X) \in \mathcal{F}(P) \wedge (t, Y) \in \mathcal{F}(Q) \wedge u \in s \mid\mid\mid t \bullet (u \downarrow \alpha, (X \cap Y) \cup Z)\} \cup \end{array}$$

$$
\begin{aligned}
& \{s : \mathit{Traces};\ X : \mathit{Refusals} \mid s \in \mathcal{D}(P \,|||\, Q) \bullet (s \downarrow \alpha, X)\} \cup \\
& \{s : \mathit{Traces};\ X : \mathit{Refusals} \mid s \in \mathcal{D}((P \,|||\, Q) \downarrow \alpha) \bullet (s, X)\} \\
= \quad & \{ \text{ definition of } \mathcal{D}((P \,|||\, Q) \downarrow \alpha) \} \\
& \{s, t, u : \mathit{Traces};\ X, Y, Z : \mathit{Refusals} \mid Z \subseteq \Sigma - \alpha \wedge \\
& \quad (s, X) \in \mathcal{F}(P) \wedge (t, Y) \in \mathcal{F}(Q) \wedge u \in s \,|||\, t \bullet (u \downarrow \alpha, (X \cap Y) \cup Z)\} \cup \\
& \{s, v : \mathit{Traces};\ X : \mathit{Refusals} \mid s \in \mathcal{D}(P \,|||\, Q) \bullet ((s \downarrow \alpha) \frown v, X)\} \\
= \quad & \{\text{property (2)}\} \\
& \{s, t, \bar{u} : \mathit{Traces};\ X, Y, Z : \mathit{Refusals} \mid Z \subseteq \Sigma - \alpha \wedge \\
& \quad (s, X) \in \mathcal{F}(P) \wedge (t, Y) \in \mathcal{F}(Q) \wedge \bar{u} \in (s \downarrow \alpha) \,|||\, (t \downarrow \alpha) \wedge \bullet \\
& \quad (\bar{u}, (X \cap Y) \cup Z)\} \cup \\
& \{s, v : \mathit{Traces};\ X : \mathit{Refusals} \mid s \in \mathcal{D}(P \,|||\, Q) \bullet ((s \downarrow \alpha) \frown v, X)\} \\
= \quad & \{ \text{ definition of } \mathcal{D}((P \,|||\, Q) \downarrow \alpha) \text{ and } (X \cup V) \cap (Y \cup W) = \\
& \quad (X \cap Y) \cup (X \cap W) \cup (V \cap Y) \cup (V \cap W) \} \\
& \{s, t, \bar{u} : \mathit{Traces};\ X, Y, V, W : \mathit{Refusals} \mid V \subseteq \Sigma - \alpha \wedge W \subseteq \Sigma - \alpha \wedge \\
& \quad (s, X) \in \mathcal{F}(P) \wedge (t, Y) \in \mathcal{F}(Q) \wedge \bar{u} \in (s \downarrow \alpha) \,|||\, (t \downarrow \alpha) \bullet \\
& \quad (\bar{u}, (X \cup V) \cap (Y \cup W)\} \cup \\
& \{s : \mathit{Traces};\ X : \mathit{Refusals} \mid s \in \mathcal{D}((P \,|||\, Q) \downarrow \alpha) \bullet (s, X)\} \\
\subseteq \quad & \{ \text{ part (ii) and definition of } \downarrow \alpha \text{ adding failures due to } \mathcal{D}(P \downarrow \alpha) \} \\
& \{\bar{s}, \bar{t}, \bar{u} : \mathit{Traces};\ \bar{X}, \bar{Y} : \mathit{Refusals} \mid \\
& \quad (\bar{s}, \bar{X}) \in \mathcal{F}(P \downarrow \alpha) \wedge (\bar{t}, \bar{Y}) \in \mathcal{F}(Q \downarrow \alpha) \wedge \bar{u} \in \bar{s} \,|||\, \bar{t} \bullet (\bar{u}, \bar{X} \cap \bar{Y}\} \cup \\
& \{s : \mathit{Traces};\ X : \mathit{Refusals} \mid s \in \mathcal{D}(P \downarrow \alpha \,|||\, Q \downarrow \alpha) \bullet (s, X)\} \\
= \quad & \{ \text{ definition of } ||| \} \\
& \mathcal{F}(P \downarrow \alpha \,|||\, Q \downarrow \alpha).
\end{aligned}
$$

Re (ii): We deduce the following chain of equations:

$$
\begin{aligned}
& \mathcal{D}((P \,|||\, Q) \downarrow \alpha) \\
= \quad & \{ \text{ definition of } \downarrow \alpha \} \\
& \{w, z : \mathit{Traces} \mid w \in \mathcal{D}(P \,|||\, Q) \bullet (w \downarrow \alpha) \frown z\} \\
= \quad & \{ \text{ definition of } ||| \} \\
& \{s, t, u, v, z : \mathit{Traces} \mid (s, \varnothing) \in \mathcal{F}(P) \wedge (t, \varnothing) \in \mathcal{F}(Q) \wedge \\
& \quad (s \in \mathcal{D}(P) \vee t \in \mathcal{D}(Q)) \wedge u \in s \,|||\, t \bullet ((u \frown v) \downarrow \alpha) \frown z\} \\
= \quad & \{ \text{ property (2) above and } \downarrow \alpha \text{ distributes over } \frown \text{ of traces } \} \\
& \{s, t, \bar{u}, \bar{v}, z : \mathit{Traces} \mid (s, \varnothing) \in \mathcal{F}(P) \wedge (t, \varnothing) \in \mathcal{F}(Q) \wedge \\
& \quad (s \in \mathcal{D}(P) \vee t \in \mathcal{D}(Q)) \wedge \bar{u} \in (s \downarrow \alpha) \,|||\, (t \downarrow \alpha) \wedge \bar{v} \in \alpha^* \bullet \bar{u} \frown \bar{v} \frown z\}
\end{aligned}
$$

$$
\begin{aligned}
= \quad & \{ \text{ definition of } \downarrow \alpha \ \} \\
& \{\bar{s}, \bar{t}, \bar{u}, \bar{v}, z : \mathit{Traces} \mid (\bar{s}, \varnothing) \in \mathcal{F}(P \downarrow \alpha) \wedge (\bar{t}, \varnothing) \in \mathcal{F}(Q \downarrow \alpha) \wedge \\
& \quad (\bar{s} \in \mathcal{D}(P \downarrow \alpha) \vee \bar{t} \in \mathcal{D}(Q \downarrow \alpha)) \wedge \bar{u} \in \bar{s} \mid\mid\mid \bar{t} \wedge \bar{v} \in \alpha^* \bullet \bar{u} \frown \bar{v} \frown z\} \\
= \quad & \{ \alpha^* \frown \Sigma^* = \Sigma^* \ \} \\
& \{\bar{s}, \bar{t}, \bar{u}, v : \mathit{Traces} \mid (\bar{s}, \varnothing) \in \mathcal{F}(P \downarrow \alpha) \wedge (\bar{t}, \varnothing) \in \mathcal{F}(Q \downarrow \alpha) \wedge \\
& \quad (\bar{s} \in \mathcal{D}(P \downarrow \alpha) \vee \bar{t} \in \mathcal{D}(Q \downarrow \alpha)) \wedge \bar{u} \in \bar{s} \mid\mid\mid \bar{t} \bullet \bar{u} \frown v\} \\
= \quad & \{ \text{ definition of } \mid\mid\mid \ \} \\
& \mathcal{D}(P \downarrow \alpha \mid\mid\mid Q \downarrow \alpha). \quad \square
\end{aligned}
$$

Furthermore, it can be shown that projection and interleaving with the chaotic process have a complementary effect.

Lemma 3. *For CSP processes A, C with $\alpha(A) \subseteq \alpha(C)$ and $N = \alpha(C) - \alpha(A)$ the following refinement relation holds in the trace model:*

$$C \downarrow \alpha(A) \mid\mid\mid \mathit{CHAOS}(N) \sqsubseteq_{\mathcal{T}} C.$$

Proof. By definition of $\sqsubseteq_{\mathcal{T}}$ we have to show the inclusion

$$\mathcal{T}(C) \subseteq \mathcal{T}(C \downarrow \alpha(A) \mid\mid\mid \mathit{CHAOS}(N)).$$

Consider a trace $s \in \mathcal{T}(C)$. Then $s \downarrow \alpha(A) \in \mathcal{T}(C \downarrow \alpha(A))$. Let t be the sequence of elements removed from s to obtain $s \downarrow \alpha(A)$. Note that $t \in \mathcal{T}(\mathit{CHAOS}(N))$ holds. Thus $s \in s \downarrow \alpha(A) \mid\mid\mid t$ and hence $s \in \mathcal{T}(C \downarrow \alpha(A) \mid\mid\mid \mathit{CHAOS}(N))$ as desired. $\square$

Lemma 4. *For CSP processes A, C with $\alpha(A) \subseteq \alpha(C)$ and $N = \alpha(C) - \alpha(A)$ the following refinement relation holds in the failure divergence model:*

$$C \downarrow \alpha(A) \mid\mid\mid \mathit{CHAOS}(N) \sqsubseteq_{\mathcal{FD}} C.$$

Proof. By definition of $\sqsubseteq_{\mathcal{T}}$ we have to show:

(i) $\mathcal{F}(C) \subseteq \mathcal{F}(C \downarrow \alpha(A) \mid\mid\mid \mathit{CHAOS}(N))$
(ii) $\mathcal{D}(C) \subseteq \mathcal{D}(C \downarrow \alpha(A) \mid\mid\mid \mathit{CHAOS}(N))$.

Re (i): Consider $(s, X) \in \mathcal{F}(C)$. Let $Z \subseteq \Sigma - \alpha(A)$. Then $(s \downarrow \alpha(A), X \cup Z) \in \mathcal{F}(C \downarrow \alpha(A))$. Let t be the sequence of elements removed from s to obtain $s \downarrow \alpha(A)$. Then $(t, X) \in \mathcal{F}(\mathit{CHAOS}(N))$. Also $s \in s \downarrow \alpha(A) \mid\mid\mid t$ and $X = (X \cup Z) \cap X$. Thus $(s, X) \in \mathcal{F}(C \downarrow \alpha(A) \mid\mid\mid \mathit{CHAOS}(N))$.

Re (ii): Consider $s \in \mathcal{D}(C)$. Then $s \downarrow \alpha(A) \in \mathcal{D}(C \downarrow \alpha(A))$ and thus $(s \downarrow \alpha(A), \varnothing) \in \mathcal{F}(C \downarrow \alpha(A))$. Let t be as above. Then $(t, \varnothing) \in \mathcal{F}(\mathit{CHAOS}(N))$ and $s \in s \downarrow \alpha(A) \mid\mid\mid t$. Hence $s \in \mathcal{D}(C \downarrow \alpha(A) \mid\mid\mid \mathit{CHAOS}(N))$ as desired. $\square$

6.2. Proofs

We wish to present a uniform proof of the theorems in Section 5 for both the trace and the failure divergence model of CSP, i.e. give a joint proof for the two theorems on

property inheritance and for those about the smallest process inheriting properties. For this purpose, let $\mathcal{M}$ stand for either $\mathcal{T}$ or $\mathcal{FD}$. Thus for CSP processes P, Q we define $P =_{\mathcal{M}} Q$ iff $\mathcal{M}(Q) = \mathcal{M}(P)$ and $P \sqsubseteq_{\mathcal{M}} Q$ iff $\mathcal{M}(Q) \subseteq \mathcal{M}(P)$ holds.

As in Section 5 we consider processes A, C with $\alpha(A) \subseteq \alpha(C)$ and $N = \alpha(C) - \alpha(A)$. The next theorem restates Theorems 1 and 3 of Section 5, but now parameterised by the model $\mathcal{M}$.

Theorem 6. *Suppose* $A \,|||\, CHAOS(N) \sqsubseteq_{\mathcal{M}} C$. *Then for all processes* P

$$P \sqsubseteq_{\mathcal{M}} A \quad \textit{implies} \quad P \downarrow \alpha(A) \sqsubseteq_{\mathcal{M}} C \downarrow \alpha(A).$$

Proof. We use the following chain of reasoning:

$$\begin{array}{ll} & P \sqsubseteq_{\mathcal{M}} A \\ \Rightarrow & \quad \{ \text{ monotonicity of } ||| \ \} \\ & P \,|||\, CHAOS(N) \sqsubseteq_{\mathcal{M}} A \,|||\, CHAOS(N) \\ \Rightarrow & \quad \{ \text{ assumption } A \,|||\, CHAOS(N) \sqsubseteq_{\mathcal{M}} C \text{ and transitivity of } \sqsubseteq_{\mathcal{M}} \} \\ & P \,|||\, CHAOS(N) \sqsubseteq_{\mathcal{M}} C \\ \Rightarrow & \quad \{ \text{ monotonicity of } \downarrow \alpha(A) \} \\ & (P \,|||\, CHAOS(N)) \downarrow \alpha(A) \sqsubseteq_{\mathcal{M}} C \downarrow \alpha(A) \\ \Rightarrow & \quad \{\downarrow \alpha(A) \text{ sub-distributes over } |||, \text{ see Lemmas 1 and 2 } \} \\ & P \downarrow \alpha(A) \,|||\, CHAOS(N) \downarrow \alpha(A) \sqsubseteq_{\mathcal{M}} C \downarrow \alpha(A) \\ \Rightarrow & \quad \{CHAOS(N) \downarrow \alpha(A) =_{\mathcal{M}} STOP \text{ because } A \cap N = \varnothing\} \\ & P \downarrow \alpha(A) \,|||\, STOP \sqsubseteq_{\mathcal{M}} C \downarrow \alpha(A) \\ \Rightarrow & \quad \{ \text{ STOP is neutral element w.r.t. } |||\} \\ & P \downarrow \alpha(A) \sqsubseteq_{\mathcal{M}} C \downarrow \alpha(A). \quad \square \end{array}$$

The next theorem restates Theorems 2 and 4 of Section 5, parameterised in the model $\mathcal{M}$.

Theorem 7. $C = A \,|||\, CHAOS(N)$ *is the w.r.t.* $\sqsubseteq_{\mathcal{M}}$ *smallest process such that for all processes* P

$$P \sqsubseteq_{\mathcal{M}} A \quad \textit{implies} \quad P \downarrow \alpha(A) \sqsubseteq_{\mathcal{M}} C \downarrow \alpha(A).$$

Proof. By Theorem 6, process C defined as $A \,|||\, CHAOS(N)$ indeed inherits all properties P of A with respect to $\alpha(A)$. The subsequent Lemma 5 proves that C is the smallest process by showing that all processes which are *not* refinements of $A \,|||\, CHAOS(N)$ cannot inherit all properties of A. $\square$

Lemma 5. *Suppose* $A \,|||\, CHAOS(N) \not\sqsubseteq_{\mathcal{M}} C$. *Then there exists some process* P *with*

$$P \sqsubseteq_{\mathcal{M}} A \quad \textit{and} \quad P \downarrow \alpha(A) \not\sqsubseteq_{\mathcal{M}} C \downarrow \alpha(A).$$

Proof. Choose $P = A$. Then obviously $P \sqsubseteq_{\mathcal{M}} A$. Suppose we have $P \downarrow \alpha(A) \sqsubseteq_{\mathcal{M}} C \downarrow \alpha(A)$. Then we use the following chain of reasoning:

$$
\begin{array}{ll}
 & P \downarrow \alpha(A) \sqsubseteq_{\mathcal{M}} C \downarrow \alpha(A) \\
\Rightarrow & \quad \{ \text{ monotonicity of } ||| \ \} \\
 & P \downarrow \alpha(A) \ ||| \ CHAOS(N) \sqsubseteq_{\mathcal{M}} C \downarrow \alpha(A) \ ||| \ CHAOS(N) \\
\Rightarrow & \quad \{C \downarrow \alpha(A) \ ||| \ CHAOS(N) \sqsubseteq_{\mathcal{M}} C, \text{ see Lemmas 3 and 4 } \} \\
 & P \downarrow \alpha(A) \ ||| \ CHAOS(N) \sqsubseteq_{\mathcal{M}} C \\
\Rightarrow & \quad \{ \text{ choice of } P \text{ and } A \downarrow \alpha(A) =_{\mathcal{M}} A \text{ and compositionality of } \mathcal{M} \text{ w.r.t. } ||| \ \} \\
 & A \ ||| \ CHAOS(N) \sqsubseteq_{\mathcal{M}} C
\end{array}
$$

This contradicts $A \ ||| \ CHAOS(N) \not\sqsubseteq_{\mathcal{M}} C$. □

7. Conclusion

In this paper we took the combined specification formalism CSP-OZ to define and study the inheritance of properties from superclasses to subclasses. Semantically, classes, instances, and systems in CSP-OZ denote processes in the standard failure divergence model of CSP. This allowed us to make full use of the well established mathematical theory of CSP. In the case of systems with finite state CSP parts and finite data types in the OZ parts the FDR model-checker for CSP can be applied to automatically verify refinement relations between CSP-OZ specifications [14] and verify subtyping relations [46].

We showed that the inheritance of safety and liveness properties requires a failure divergence subtype relationship between super- and subclass. Failure divergence subtyping is a strong requirement for subclasses to satisfy, which may not always be achievable. As future work it would be interesting to study weaker conditions under which properties are inherited. These conditions might be parameterised by the property of interest, i.e. for a specific property to be inherited specific conditions have to be checked.

Related work. A number of other combinations of process algebra with formalisms for describing data exist today. A comparison of approaches for combining Z (or B) with a process algebra can be found in [11]. Such integrations include Timed CSP and Object-Z (TCOZ) [28], B and CSP [4] and Z and CCS [43,16]. Closest to the combination CSP-OZ is Object-Z/CSP due to Smith [38]. There CSP operators serve to combine Object-Z classes and instances. Thus to Object-Z classes a semantics in the failures divergence model of CSP is assigned just as is done here for CSP-OZ. This semantics is obtained as an abstraction of the more detailed history semantics of Object-Z [37]. In contrast to CSP-OZ there is no CSP-part *inside* classes. As we have seen in the example, the CSP part is convenient for specifying sequencing constraints on the communications events. Both CSP-OZ and Object-Z/CSP have been extended to deal with real-time [21,40].

The issue of inheritance of properties to subtypes has been treated by van der Aalst and Basten [44]. They deal with net-specific properties like safety (of nets), deadlock freedom and free choice.

Leavens and Weihl [25] show how to verify object-oriented *programs* using a technique called "supertype abstraction". This technique is based on the idea that subtypes need not to be re-verified once a property has been proven for their supertypes. In their study they have to take particular care about *aliasing* since in object-oriented programs several references may point to the same object, and thus an object may be manipulated in several ways. Subtyping for object-oriented programs has to avoid references which are local to the supertype but accessible in the subtype.

Preservation of properties is also an issue in transformations within the language UNITY proposed by Chandy and Misra [5]. The *superposition* operator in UNITY is a form of parallel composition which requires that the new part does not make assignments to underlying (old) variables. This is close to the non-modification condition we used in Theorem 5. Superposition preserves all properties of the original program.

Acknowledgement

We thank the referee for detailed and helpful comments on this paper.

References

[1] P. America, Designing an object-oriented programming language with behavioural subtyping, in: J.W. de Bakker, W.P. de Roever, G. Rozenberg (Eds.), REX Workshop: Foundations of Object-Oriented Languages, LNCS, vol. 489, Springer, 1991.
[2] C. Bolton, J. Davies, Refinement in Object-Z and CSP, in: M. Butler, L. Petre, K. Sere (Eds.), Integrated Formal Methods, IFM 2002, Lecture Notes in Computer Science, vol. 2335, Springer-Verlag, 2002, pp. 225–244.
[3] S.D. Brookes, C.A.R. Hoare, A.W. Roscoe, A theory of communicating sequential processes, Journal of the ACM 31 (1984) 560–599.
[4] M. Butler, csp2B: A practical approach to combining CSP and B, in: J. Wing, J. Woodcock, J. Davies (Eds.), FM'99: Formal Methods, Lecture Notes in Computer Science, vol. 1708, Springer, 1999, pp. 490–508.
[5] K.M. Chandy, J. Misra, Parallel Program Design—A Foundation, Addison-Wesley, 1988.
[6] B.T. Denvir, J. Oliveira, N. Plat, The cash-point (ATM) 'Problem', Formal Aspects of Computing 12 (4) (2000) 211–215.
[7] J. Derrick, E.A. Boiten, Refinement in Z and Object-Z, Springer-Verlag, 2001.
[8] K.K. Dhara, G.T. Leavens, Forcing behavioral subtyping through specification inheritance, in: Proceedings of the 18th International Conference on Software Engineering, March, Berlin, Germany, IEEE Computer Society Press, 1996, pp. 258–267.
[9] R. Duke, G. Rose, G. Smith, Object-Z: A specification language advocated for the description of standards, Computer Standards and Interfaces 17 (1995) 511–533.
[10] C. Fischer, CSP-OZ: A combination of Object-Z and CSP, in: H. Bowman, J. Derrick (Eds.), Formal Methods for Open Object-Based Distributed Systems, FMOODS'97, vol. 2, Chapman & Hall, 1997, pp. 423–438.
[11] C. Fischer, How to combine Z with a process algebra, in: J. Bowen, A. Fett, M. Hinchey (Eds.), ZUM'98 The Z Formal Specification Notation, LNCS, vol. 1493, Springer, 1998, pp. 5–23.
[12] C. Fischer, Combination and Implementation of Processes and Data: From CSP-OZ to Java. Ph.D. Thesis, Bericht Nr. 2/2000, University of Oldenburg, April 2000.
[13] C. Fischer, G. Smith, Combining CSP and Object-Z: Finite or infinite trace-semantics? in: T. Mizuno, N. Shiratori, T. Higashino, A. Togashi (Eds.), Proceedings of FORTE/PSTV'97, Chapmann & Hall, 1997, pp. 503–518.
[14] C. Fischer, H. Wehrheim, Model-checking CSP-OZ specifications with FDR, in: K. Araki, A. Galloway, K. Taguchi (Eds.), Integrated Formal Methods, Springer, 1999, pp. 315–334.

[15] C. Fischer, H. Wehrheim, Behavioural subtyping relations for object-oriented formalisms, in: T. Rus (Ed.), AMAST 2000: International Conference on Algebraic Methodology and Software Technology, Lecture Notes in Computer Science, vol. 1816, Springer, 2000, pp. 469–483.
[16] A.J. Galloway, W. Stoddart, An operational semantics for ZCCS, in: M. Hinchey, S. Liu (Eds.), Int. Conf. of Formal Engineering Methods, ICFEM, IEEE, 1997.
[17] J.V. Guttag, J.J. Horning, J.M. Wing, The larch family of specification languages, IEEE Software 2 (5) (1985) 24–36.
[18] J. Hatcliff, M. Dwyer, Using the Bandera tool set to model-check properties of concurrent Java software, in: K.G. Larsen (Ed.), CONCUR 2001, LNCS, Springer, 2001.
[19] C.A.R. Hoare, Communicating sequential processes, CACM 21 (1978) 666–677.
[20] C.A.R. Hoare, Communicating Sequential Processes, Prentice Hall, 1985.
[21] J. Hoenicke, E.-R. Olderog, Combining specification techniques for processes, data and time, in: M. Butler, L. Petre, K. Sere (Eds.), Integrated Formal Methods, IFM 2002, LNCS, vol. 2335, Springer, 2002, pp. 245–266.
[22] M. Huisman, B. Jacobs, Java program verification via a Hoare logic with abrupt termination, in: T. Maibaum (Ed.), Fundamental Approaches to Software Engineering, FASE 2000, LNCS, vol. 1783, Springer, 2000, pp. 284–303.
[23] Kolyang, HOL-Z—An integrated formal support environment for Z in Isabelle/HOL, Ph.D. Thesis, Univ. Bremen, 1997, Shaker Verlag, Aachen, 1999.
[24] D. Latella, I. Majzik, M. Massink, Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker, Formal Aspects of Computing 11 (1999) 430–445.
[25] G.T. Leavens, W.E. Weihl, Specification and verification of object-oriented programs using supertype abstraction, Acta Informatica 32 (1995) 705–778.
[26] K.R.M. Leino, Extended static checking: A ten-year perspective, in: R. Wilhelm (Ed.), Informatics—10 Years Back, 10 Years Ahead, LNCS, vol. 2000, Springer, 2001, pp. 157–175.
[27] B. Liskov, J. Wing, A behavioural notion of subtyping, ACM Transactions on Programming Languages and Systems 16 (6) (1994) 1811–1841.
[28] B.P. Mahony, J.S. Dong, Blending Object-Z and timed CSP: an introduction to TCOZ, in: The 20th International Conference on Software Engineering, ICSE'98, April, IEEE Computer Society Press, 1998, pp. 95–104.
[29] O. Nierstrasz, Regular types for active objects, in: O. Nierstrasz, D. Tsichritzis (Eds.), Object-oriented software composition, Prentice Hall, 1995, pp. 99–121.
[30] E.-R. Olderog, C.A.R. Hoare, Specification-oriented semantics for communicating processes, Acta Informatica 23 (1986) 9–66.
[31] E.-R. Olderog, H. Wehrheim, Specification and inheritance in CSP-OZ, in: F.S. de Boer, M. Bonsague, W.P. de Roever (Eds.), Formal Methods for Components and Objects, LNCS, vol. 2852, Springer, 2003, pp. 361–379.
[32] A. Poetzsch-Heffter, J. Meyer, Interactive verification environments for object-oriented languages, Journal of Universal Computer Science 5 (3) (1999) 208–225.
[33] A.W. Roscoe, Model-checking CSP, in: A.W. Roscoe (Ed.), A Classical Mind—Essays in Honour of C.A.R. Hoare, Prentice-Hall, 1994, pp. 353–378.
[34] A.W. Roscoe, The Theory and Practice of Concurrency, Prentice-Hall, 1997.
[35] M. Saaltink, The Z/EVES system, in: J. Bowen, M. Hinchey, D. Till (Eds.), ZUM'97, LNCS, vol. 1212, Springer, 1997, pp. 72–88.
[36] T. Schäfer, A. Knapp, S. Merz, Model checking UML state machines and collaborations, in: Workshop on Software Model Checking, ENTCS, vol. 55, 2001.
[37] G. Smith, A fully abstract semantics of classes for Object-Z, Formal Aspects of Computing 7 (1995) 289–313.
[38] G. Smith, A semantic integration of Object-Z and CSP for the specification of concurrent systems, in: J. Fitsgerald, C.B. Jones, P. Lucas (Eds.), Formal Methods Europe, FME'97, LNCS, vol. 1313, Springer, 1997, pp. 62–81.
[39] G. Smith, The Object-Z Specification Language, Kluwer Academic Publisher, 2000.

[40] G. Smith, An integration of real-time Object-Z and CSP for specifying concurrent real-time systems, in: M. Butler, L. Petre, K. Sere (Eds.), Integrated Formal Methods, IFM 2002, LNCS, vol. 2335, Springer, 2002, pp. 267–285.
[41] G. Smith, F. Kammüller, T. Santen, Encoding Object-Z in Isabelle/HOL, in: D. Bert, J.P. Bowen, M.C. Henson, K. Robinson (Eds.), ZB 2002: Formal Specification and Development in Z and B, LNCS, vol. 2272, Springer, 2002, pp. 82–99.
[42] J.M. Spivey, The Z Notation: A Reference Manual, 2nd edition, Prentice-Hall International Series in Computer Science, 1992.
[43] K. Taguchi, K. Araki, Specifying concurrent systems by Z + CCS, in: International Symposium on Future Software Technology, ISFST, 1997, pp. 101–108.
[44] W.M.P. van der Aalst, T. Basten, Inheritance of Workflows—An approach to tackling problems related to change, Theoretical Computer Science 270 (1–2) (2002) 125–203.
[45] H. Wehrheim, Behavioural subtyping in object-oriented specification formalisms, University of Oldenburg, Habilitation Thesis, 2002.
[46] H. Wehrheim, Checking behavioural subtypes via refinement, in: A. Rensink, B. Jacobs (Eds.), FMOODS 2002: Formal Methods for Open Object-Based Distributed Systems, Kluwer, 2002, pp. 79–93.
[47] H. Wehrheim, Behavioral subtyping relations for active objects, Formal Methods in System Design 23 (2003) 143–170.
[48] J. Woodcock, J. Davies, Using Z—Specification, Refinement, and Proof, Prentice-Hall, 1996.