# Compositional Modelling and Reasoning in an Institution for Processes and Data

Liam O'Reilly[1], Till Mossakowski[2], and Markus Roggenbach[1]

[1] Swansea University, Wales, UK
[2] DFKI GmbH Bremen, Bremen, Germany

**Abstract.** The language Csp-Casl combines specifications of data and processes. We give an institution based semantics to Csp-Casl that allows us to re-use the institution independent structuring mechanisms of Casl. Furthermore, we extend Csp-Casl with a notion of refinement that reconciles the differing philosophies behind the refinement notions for Csp and Casl. We develop a compositional proof calculus for refinement along the Casl structuring mechanisms, and demonstrate that compositional proof techniques along parallel process composition from the context of Csp lifts to structured Csp-Casl specifications.

## 1 Introduction

Distributed computer applications like flight booking systems, web services, and electronic payment systems such as the EP2 standard [1] involve the parallel processing of data. Consequently, these systems exhibit concurrent aspects (e.g., deadlock freedom) as well as data aspects (e.g., functional correctness). Often, these aspects depend on each other. In [20], we present the language Csp-Casl, which is tailored to the specification of distributed systems. Csp-Casl integrates the process algebra Csp [10, 21] with the algebraic specification language Casl [15].

In [8] we apply Csp-Casl to the EP2 standard and demonstrate that Csp-Casl can deal with problems of industrial strength. Interestingly enough, Csp alone is not expressive enough to model the EP2 standard: The abstract system descriptions included in this standard require loose semantics of data. However, the exercise in [8] also demonstrates the need to enrich Csp-Casl by means for specification in the large: While the Casl structuring mechanisms are available for data to be plugged into a Csp-Casl specification, this has yet no counterpart on the process side.

Based on an institution for Csp [14], here we extend this language by loose processes and give it an institution-based semantics. The institutional setting [9] allows for specifications with loosely specified data and process parts. Moreover, the institution independent structuring mechanisms of Casl can be applied in the process algebraic setting in a methodologically meaningful way.

Furthermore, we study refinement in the context of Csp-Casl. Refinement in Casl is usually reduced to simple model class inclusion, given the power of the

T. Mossakowski and H.-J. Kreowski (Eds.): WADT 2010, LNCS 7137, pp. 251–269, 2012.

CASL structuring mechanisms that can be used to massage the involved specifications if needed. We show that a similar approach can be used for capturing CSP's traditional notion of refinement also in the setting of loose semantics. Moreover, we show that reasoning about refinements can be done in a modular way, using the CASL structuring mechanisms.

CSP-CASL also inherits from the process algebraic side: For CSP, [18] presents a compositional approach for deadlock analysis on networks of processes. We lift this technique to CSP-CASL, and show by means of an extended example, how to use it in combination with the structuring constructs inherited from CASL.

To the best of our knowledge, we were the first to suggest loose process specifications in [14]. Here, we combine this idea with loose data specifications. Accordingly, our notion of refinement for loose data and processes is new as well. Other approaches of combining data and processes, e.g., CSP-M [23], CSP-Z [7], and CSP-OZ [24], use tight semantics of both data and processes and provide only limited structuring. The WRIGHT architectural description language [2] allows reasoning on typed processes for a sublanguage of CSP; semantically, it is restricted to a single CSP model. Moreover, WRIGHT does not cover data refinement. Temporal logics offer a declarative approach to the specification of reactive behaviour. Here, [25] studies structuring of reactive systems using CASL architectural specifications over an institution of transition systems and $CTL^*$ formulae. This again differs from our work, as we consider structured specification with loose semantics (classes of models), whereas architectural specifications focus on the structuring of individual models. In other reactive CASL extensions, e.g., MODALCASL [12] or CASL-LTL [19], the concept of refinement and its interaction with structuring has not been studied yet.

Our paper is organised as follows: In Section 2 we motivate our notion of "loose processes". Then we develop, to some extent, institutions for CSP-CASL: one institution for each of the main CSP models, namely the CSP traces model, the CSP failures/divergences model, and the CSP stable failures model. Section 4 defines CSP-CASL refinement and gives compositional proof rules along the CASL structuring mechanisms. Then we discuss how to lift a compositional deadlock analysis rule from the CSP context to CSP-CASL. We conclude the paper with an extended example.

We assume that the reader is familiar with CSP ([10, 21] provide introductions) and with CASL ([4] is a gentle introduction). Moreover, we use the notion of institutions [9] as a formalisation of the notion of logical system. The reader unfamiliar with institutions should be able to understand most parts of this paper when replacing the word "institution" by "logical system".

## 2 Loose Process Semantics

CSP-CASL [20] is a novel specification language which combines *processes* written in CSP [10, 21] with the specification of *data types* in CASL [15]. The general idea is to describe reactive systems in the form of processes based on CSP operators, where the communications of these processes are the values of data types, which

are loosely specified in CASL. All standard CSP operators are available, such as multiple prefix, the various parallel operators, operators for non-deterministic choice, and communication over channels. Concerning CASL features, the full language is available to specify data types, namely many-sorted first order logic with sort-generation constraints, partiality, and subsorting.

CSP-CASL supports the three main CSP semantics: The traces model $\mathcal{T}$, in which one can verify safety properties; the failures/divergences model $\mathcal{N}$, which allows one to study the phenomenon of livelock, i.e., the possibility that the system can indefinitely engage in internal actions only; and the stable failures model $\mathcal{F}$, which is best suited for deadlock analysis. The traces model $\mathcal{T}$ records only the possible traces of a process; the failures/divergences model records two different behaviours: The failures – i.e., action sets which a process can refuse after executing a trace – and the divergences – i.e., traces that lead to a livelock; the stable failures model, finally, records two behaviours: The system traces exactly like the traces model, and the failures for "stable" states, i.e., states which can't perform an internal action. The main means of verification in CSP is to prove that one process, say $P$, refines to another one, say $Q$, in signs $P \sqsubseteq Q$. Each CSP model gives rise to one notion of refinement. Here, the following relations have been established: $\sqsubseteq_{\mathcal{N}} \subsetneq \sqsubseteq_{\mathcal{T}}$, $\sqsubseteq_{\mathcal{F}} \subsetneq \sqsubseteq_{\mathcal{T}}$, $\sqsubseteq_{\mathcal{N}} \not\subseteq \sqsubseteq_{\mathcal{F}}$, and $\sqsubseteq_{\mathcal{F}} \not\subseteq \sqsubseteq_{\mathcal{N}}$, see [21].

In this paper, we extend the setting of CSP-CASL as defined in [20] by adding loose semantics for processes, following the ideas of [14]. Loose process semantics offers advantages in terms of methodology, furthermore, it is required for generic specifications and instantiation.

For the methodological aspect, consider the specification ARCH_CUSTOMER of the customer of an electronic shop, see Figure 1 – in the context of our example, to be discussed in more detail in Section 6 – we call this the "architectural level". The data part written in CASL provides a type system, namely that *LoginReq* ("Login Request") and *Logout* are subsorts of *D_C* ("Data Customer"), which comprises of all data the customer can deal with. The customer communicates to the outside world over a channel *C_C* ("Channel Customer"), which allows for messages of type *D_C*. The suffix *_def* on sort names excludes the "error" element of the sort, i.e., we are specifying the system under the assumption that only valid messages are exchanged.[1] In the process part, the customer's behaviour is described in terms of several processes, devoted to different activities. The purpose of the architectural level is to describe how to combine these activities in order to describe the customer. The detailed description of such an activity, e.g., *Customer_GoodLogin*, however, is postponed to a later design step. We only state that there is a process *Customer_GoodLogin*, whose behaviour is underspecified, i.e., in semantical terms it is "loose".

With such loose specifications available for the customer, warehouse, payment system, and the coordinator, we can model the whole shop as their parallel composition over various channels, see the process part of the specification ARCH_SHOP. Here, the specifications ARCH_CUSTOMER, ARCH_WAREHOUSE,

[1] For simplicity we refrain from error handling.

ARCH_PAYMENTSYSTEM and ARCH_COORDINATOR serve as parameters in a generic construction. They provide the names and properties of data and processes involved. But what instances do we want to allow? Obviously, any refinement of these parameters shall be possible. To this end, we define an operator *RefCl* – to be discussed in detail in Section 4 – which closes the model class of a CSP-CASL specification under refinement.

Loose or underspecified processes differ from non-deterministic processes in CSP. The process $P = a \rightarrow Stop \sqcap b \rightarrow Stop$ is non deterministic. For this equation, there is only *one* denotation possible for $P$ which makes the equation true. In the traces model $\mathcal{T}$, e.g., this is the interpretation $I(P) = \{\langle\rangle, \langle a\rangle, \langle b\rangle\}$. Specifying a loose process $Q$ by saying $Q$ shall be any process that refines to $P$, written $Q \sqsubseteq P$, however, leads to infinitely many different possible denotations of $Q$. In the traces model $\mathcal{T}$, e.g., we have the interpretations $J(Q) = I(P)$ and $K(Q) = \{\langle\rangle, \langle a\rangle, \langle a, a\rangle, \langle b\rangle\}$. Note that this example also demonstrates that the set of interpretations of loose processes is not necessarily refinement closed: $\{\langle\rangle\}$ is not a possible interpretation for $Q$, however it is an element of every refinement closed set in $\mathcal{T}$.

**spec** ARCH_CUSTOMER =
  **data sorts** *LoginReq*, *Logout* < *D_C*
  **channel** *C_C* : *D_C*
  **process** *Customer* : *C_C* ; *Customer_GoodLogin* : *C_C* ;
    *Customer_BadLogin* : *C_C* ; *Customer_AddItem* : *C_C* ;
    *Customer_Body* : *C_C* ; *Customer_Quit* : *C_C* ; ...
    *Customer* = *C_C* ! *x* :: *LoginReq_def* →
      (*Customer_GoodLogin* ; *Customer_Body* □
        *Customer_BadLogin* ; *Customer*)
    *Customer_Quit* = *C_C* ! *x* :: *Logout_def* → *SKIP*
    *Customer_Body* = *Customer_AddItem* ⊓ ... ⊓ *Customer_Quit*
**end**

**spec** ARCH_SHOP [*RefCl*(ARCH_CUSTOMER)] [*RefCl*(ARCH_WAREHOUSE)]
  [*RefCl*(ARCH_PAYMENTSYSTEM)] [*RefCl*(ARCH_COORDINATOR)] =
  **process** *System* : *C_C*, *C_W*, *C_PS* ;
    *System* = *Coordinator* ⟦ *C_C*, *C_W*, *C_PS* || *C_C*, *C_W*, *C_PS* ⟧
      (*Customer* ⟦ *C_C* || *C_W*, *C_PS* ⟧
        (*Warehouse* ⟦ *C_W* || *C_PS* ⟧ *PaymentSystem*))
**end**

**Fig. 1.** Selections of CSP-CASL specifications of our online shop example

## 3 CSP-CASL Institutions for Different CSP Models

In order to give a precise semantics to (possibly structured) CSP-CASL specifications, we formalise CSP-CASL as an institution [9]; to be more precise: three institutions – one for each of the main CSP models, namely: the CSP traces model, the CSP failures/divergences model, and the CSP stable failures model.

These institutions share the notions of signatures and sentences. Their respective model categories and satisfaction relations are defined following a common scheme. We only sketch the institutions, for full details see [16]. The institutions for CSP-CASL are naturally based on institutions for CASL [15] and for CSP [14], using the ideas for the CSP-CASL semantics [20] for the combination.

### 3.1 Signatures

A CSP-CASL signature $\Sigma_{CC}$ is a pair $\Sigma_{CC} = (\Sigma_{Data}, \Sigma_{Proc})$ where:

- $\Sigma_{Data} = (S, TF, PF, P, \leq)$ is a subsorted first-order signature consisting of a set of sort symbols $S$, a set of total functions symbols $TF$, a set of partial function symbols $PF$, a set of predicate symbols $P$, and a reflexive and transitive subsort relation $\leq \subseteq S \times S$ – see [15] for details – where the set of sorts $S$ is finite and the subsort relation has local top elements, i.e., if $u, u' \geq s$ then there exists $t \in S$ with $t \geq u, u'$, see [20].
- $\Sigma_{Proc} = (N_{w,comms})_{w \in S^*, comms \in S_\downarrow}$ is a family of finite sets of process names. Such a process name $n$ is typed in the sort symbols $S$ of the data signature part:
  - a string $w = \langle s_1, \ldots, s_k \rangle$, $s_i \in S$ for $1 \leq i \leq k$, $k \geq 0$, which is $n$'s parameter type. A process name without parameters has the empty sequence $\langle\rangle$ as its parameter type.
  - a set $comms \subseteq S$ which collects all types of events in which the process $n$ can possibly engage in. We require the set $comms$ to be downward closed under the subsort relation, i.e., $comms \in S_\downarrow = \{X \subseteq S \mid X = \downarrow X\}$, where $\downarrow X = \{y \in S \mid \exists x \in X : y \leq x\}$ for $X \subseteq S$.

Given CSP-CASL signatures $\Sigma_{CC} = (\Sigma_{Data}, \Sigma_{Proc})$, $\Sigma'_{CC} = (\Sigma'_{Data}, \Sigma'_{Proc})$, with $S$ as the sort set of $\Sigma_{Data}$ and $S'$ as the sort set of $\Sigma'_{Data}$, a CSP-CASL signature morphism is a pair $\theta = (\sigma, \nu) : \Sigma_{CC} \to \Sigma'_{CC}$ where:

- $\sigma : \Sigma_{Data} \to \Sigma'_{Data}$ is a CASL signature morphism for which the following additionally hold:
  **refl** $\sigma^S(s_1) \leq_{S'} \sigma^S(s_2)$ implies $s_1 \leq_S s_2$ for all $s_1, s_2 \in S$ (reflection of the subsort relation), and
  **weak non-extension** $\sigma^S(s_1) \leq_{S'} u'$ and $\sigma^S(s_2) \leq_{S'} u'$ implies that there exists a sort $t \in S$ with $s_1 \leq_S t$, $s_2 \leq_S t$ and $\sigma^S(t) \leq_{S'} u'$.[2]
- $\nu = (\nu_{w,comms})_{w \in S^*, comms \in S_\downarrow}$ is a family of functions such that $\nu_{w,comms} : N_{w,comms} \to \cup_{comms' \in (\downarrow(\sigma(comms)))_\downarrow} N'_{\sigma(w),comms'}$ is a mapping of process names. Another way to express this is that a process name $n \in N_{w,comms}$ is mapped to $\nu_{w,comms}(n) = n'$, where $n' \in N'_{\sigma(w),comms'}$ and $\forall y \in comms' : \exists x \in comms : y \leq_{S'} \sigma(x)$ (*"the target is dominated by the source"*). We also write $\nu(n : w, comms) = n' : \sigma(w), comms'$.

[2] [20] works with the condition 'non-extension'. One can show however that the results of [20] also hold with the more liberal notion that we use here. The difference from the original version is the more liberal choice of sort $t$ (originally, we have required $t$ to be a pre-image of $u'$). Further note that for $s_1 = s_2$, the condition trivially holds.

The conditions on process translations ensure that both the parameter types as well as the communication set, are translated with the signature morphism $\sigma$ of the data part. While preserving the parameter structure, the communication set is allowed to "shrink". This "non-expansion" of the communication sets – see also [14] – guarantees that the reduct functor is defined and that the satisfaction condition holds. In [16] we demonstrate that our notion of a signature morphisms is quite liberal. All morphisms arising in our shop example fulfil these conditions.

### 3.2 Sentences

Sentences are either data or process sentences. A $\Sigma_{CC}$-data sentence is a CASL sentence over $(S, TF, PF, P, \leq)$. A $\Sigma_{CC}$-process sentence is a process definition

$$n(x_1, \ldots, x_k) = pt$$

where $n \in N_{\langle s_1,\ldots,s_k\rangle,comms}$, $x_i$ are global variables of type $s_i$, $1 \leq i \leq k$, and $pt$ is a process term such that $sorts(pt) \subseteq comms$, i.e., the process term $pt$ communicates only in events which are allowed for $n$. For further details see [16].

### 3.3 The Alphabet Construction

The alphabet construction takes a data (i.e., CASL) model and uses its elements as alphabet letters, which then form the alphabet for CSP. CSP-CASL's alphabet construction takes the subsort structure into account in order to determine whether two events are equal or not. More precisely, given a CASL model $M$, its corresponding alphabet

$$Alph(M) = (\biguplus_{s \in S} M_s \cup \{\bot_s\})/_{\sim_M}$$

is constructed by taking the disjoint union of all its carrier sets extended by a bottom element $\bot$, but identifying carriers along subsort injections (this is captured by the equivalence relation $\sim_M$). This map *Alph* extends to a functor from the model category to the category Set.

Given a CASL model $M$, we use the shorthand $M_\bot$ for the totalised version of $M$, i.e., carrier sets include a bottom element $M_\bot(s) = M(s) \cup \{\bot_s\}$ and the interpretation of function and predicate symbols is strictly extended. Given a sort symbol $s$, a CASL model $M$, and $x \in M_\bot(s)$ we write $\overline{x}^s_M$ to denote the alphabet element $[(s, x)]_{\sim_M}$. Further more we lift this notation to sorts, namely $\overline{s}_M = \{\overline{x}^s_M \mid x \in M_\bot(s)\} \subseteq Alph(M)$ for the set of communications that can arise from the sort $s$ in the model $M$. Finally, given a set of sorts $X$, we write $\overline{X}_M = \bigcup_{s \in X} \overline{s}_M$. We drop the subscripts $M$ and superscripts $s$ when clear from the context.

### 3.4 Models and Satisfaction

A CSP-CASL model consists of a data (i.e., CASL) model and a collection of interpretations for processes. Concerning the interpretation of processes, let $\mathcal{D}(A)$

be a CSP domain constructed relatively to some alphabet of communications $A$. Examples of such domains $\mathcal{D}(A)$ include $\mathcal{T}(A)$ of the CSP traces model, $\mathcal{N}(A)$ of the CSP failures/divergences model, and $\mathcal{F}(A)$ of the CSP stable failures model, see [21] for details. Each of these domains gives rise to a different institution. Actually, $\mathcal{D}$ extends to an endofunctor on the category Set.

Given a CSP-CASL signature $\Sigma_{CC} = (\Sigma_{Data}, \Sigma_{Proc})$, a $\Sigma_{CC}$-*model* is a pair $(M, I)$, where $M$ is a CASL model for $\Sigma_{Data}$ and $I$ gives type correct interpretations of the process signature in the CSP domain $\mathcal{D}(Alph(M))$. All CSP models describe, which traces a process can execute. In the following we denote these traces with the function $cTr$.[3] Type correctness of $(M, I)$ requires that the interpretation map $I$ applied to a process name $n \in N_{\langle s_1,\ldots,s_k\rangle,comms}$ for all parameters $a_i \in \overline{s_i}$, $1 \leq i \leq k$, yields an interpretation with $cTr(I(n(a_1, \ldots, a_k))) \in \mathcal{T}(\overline{comms})$. It is this type correctness condition which allows us to define the reduct functor and to prove the satisfaction condition.

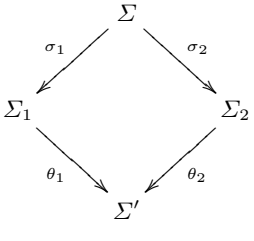
Satisfaction of data sentences w.r.t. a $\Sigma_{CC}$-model is inherited from CASL. Satisfaction of a process sentence $n(x_1, \ldots, x_k) = pt$ over signature $\Sigma_{CC}$ and global variable system $X_G$ with respect to a $\Sigma_{CC}$-model $(M, I)$ is defined as follows:

$$(M, I) \models_{\Sigma_{CC}} (n(x_1, \ldots, x_k) = pt)$$
$$\text{if and only if}$$
$$\text{for all variable valuations } \mu_G : X_G \to M_\perp :$$
$$I(n(\overline{\mu_G(x_1)}_M, \ldots, \overline{\mu_G(x_k)}_M)) = [\![[\![pt]\!]_{(M,I),\mu_G,\emptyset}]\!]_{\mathcal{D}}.$$

Here, $[\![pt]\!]_{(M,I),\mu_G,\mu_L}$ is the evaluation of process term $pt$ according to CASL with respect to model $(M, I)$ and global and local variable valuations $\mu_G$ and $\mu_L$. $[\![pt']\!]_{\mathcal{D}}$ is the denotation of process term $pt'$ in the CSP domain $\mathcal{D}$. For further details and also the definition and discussion of model morphisms see [16].

### 3.5 Pushouts and Amalgamation

The existence of pushouts and amalgamation properties shows that an institution has good modularity properties. Amalgamation is a major technical assumption in the study of specification semantics [6, 22]. An institution is said to be *semi-exact*, if for any pushout of signatures




[3] The controlled traces are the traces as given as denotations in the traces model – in $\mathcal{F}$, they are directly given, in $\mathcal{N}$, they can be computed out of the divergences and failures.

any pair $(M_1, M_2) \in \mathbf{Mod}(\Sigma_1) \times \mathbf{Mod}(\Sigma_2)$ that is *compatible*, in the sense that $M_1$ and $M_2$ reduce to the same $\Sigma$-model, can be *amalgamated* to a unique $\Sigma'$-model $M$ (i.e., there exists a unique $M \in \mathbf{Mod}(\Sigma')$ that reduces to $M_1$ and $M_2$, respectively), and similarly for model morphisms.

**Proposition 1.** CSP-CASL *signature morphisms between signatures with acyclic subsort relations are injective on sorts. Thus, CspCaslSig does not have pushouts.*

As in [14], there is a way out: Let $CspCaslSig^{plain}$ be $CspCaslSig$ with the reflection and weak non-extension restriction dropped. Then we have:

**Proposition 2.** $CspCaslSig^{plain}$ *has pushouts, and any such pushout of a span in CspCaslSig actually is a square in CspCaslSig (although not a pushout in CspCaslSig).*

Pushouts in $CspCaslSig^{plain}$ give us an amalgamation property:

**Proposition 3.** $CspCaslSig^{plain}$*-pushouts of CspCaslSig-morphisms have the semi-exactness property for the traces model and the stable failures model.*

In fact, this result generalises easily to multiple pushouts. Moreover, the initial (i.e., empty) signature has the terminal model category. Since all colimits can be formed by the initial object and multiple pushouts, this shows that we even have exactness (when colimits are taken in $CspCaslSig^{plain}$).

Altogether, Proposition 3 shows that CASL-style parameterisation, CASL architectural specifications and much more also work for CSP-CASL.

### 3.6 CSP-CASL with Channels

We often use channels in CSP-CASL. This leads to further institutions, with extended notions of signatures and sentences. Most prominently, the notion of a signature is extended by a third component $C$:

$$(\Sigma_{Data}, C, \Sigma_{Proc})$$

Here, $C$ is a finite set of names typed by non-empty lists over $S$. We require $C$ to be closed under the subsort relation[4] $\leq^*$ i.e., if $c : \langle s_1, \ldots, s_k \rangle \in C$ and $\langle u_1, \ldots, u_k \rangle \leq^* \langle s_1, \ldots, s_k \rangle$ then $c : \langle u_1, \ldots, u_k \rangle \in C$.

CSP-CASL with channels can be reduced to CSP-CASL (without channels) as follows: each CSP-CASL signature with a channel component is translated to a CSP-CASL theory $\Phi(\Sigma)$, where each channel is coded as a new sort (isomorphic to the sort of the channel) and each CSP-CASL $\Sigma$-sentence $\varphi$ is translated to a CSP-CASL $\Phi(\Sigma)$-sentence $\alpha(\varphi)$ by reducing channel communication to ordinary communication using the new channel sorts. Models and satisfaction can then be easily borrowed from CSP-CASL by letting $\mathbf{Mod}^{CC}(\Sigma) := \mathbf{Mod}^{CC}(\Phi(\Sigma))$ and $M \models^{CC}_{\Sigma} \varphi$ iff $M \models^{CC}_{\Phi(\Sigma)} \alpha(\varphi)$. This is an instance of borrowing logical structure in the sense of [5].

In the rest of the paper we use the term CSP-CASL to represent CSP-CASL with channels.

[4] $\leq^*$ stands for the pointwise extension of $\leq$ to strings of sorts.

## 4 Refinement and a Structured Proof Calculus

Refinement allows us to develop systems in a stepwise manner. Here we discuss refinement for CSP-CASL as well as its proof calculus on structured specifications.

### 4.1 Refinement

CSP has a notion of refinement between individual processes, e.g., in the traces model, $pt \sqsubseteq pt'$ means that $pt'$ has fewer traces than $pt$, i.e., $traces(pt') \subseteq traces(pt)$. In the context of this paper we write $pt \sqsubseteq pt'$ for $pt \sqsubseteq_{\mathcal{D}} pt'$ if the specific choice of $\mathcal{D} \in \{\mathcal{T}, \mathcal{N}, \mathcal{F}\}$ does not matter. Similarly, the CASL family of languages uses model class inclusion as the simplest notion of refinement [3]: $SP_1 \rightsquigarrow SP_2$ if $SP_2$ has fewer models than $SP_1$, i.e., $\mathbf{Mod}(SP_2) \subseteq \mathbf{Mod}(SP_1)$. To cater for renaming, this notion can be extended by a signature morphism $\sigma$. In this case one defines $SP_1 \rightsquigarrow^{\sigma} SP_2$ if the reduct of $SP_2$ has fewer models than $SP_1$, i.e., $\mathbf{Mod}(SP_2)|_{\sigma} \subseteq \mathbf{Mod}(SP_1)$. When combining these worlds through institution theory, one has to recognise that these two refinement notions follow different ideas: While CSP refinement talks about refinement of individual models, CASL refinement talks about refinement of model *classes*.

This should become clear with the following notion: A CSP-CASL specification $SP$ is *single-valued*, if there is no looseness in the processes, that is, any two $SP$-models with the same data part coincide. Now, traditional CSP refinement is about refinement between different single-valued process specifications – reducing the amount of internal non-determinism – whereas model class inclusion mainly captures different degrees of *looseness* of specifications.

How can we reconcile these two worlds? Here, we want to capture different degrees of looseness not only for data, but also for processes! Hence we adopt the model class inclusion notion of refinement, applied to the CSP-CASL institution. However, in order to capture CSP refinement between different single-valued processes (which alone, would obviously never lead to model class inclusion), we also provide a notion of refinement closure (and here, "refinement" is meant in the CSP sense, not in the model class inclusion sense).

Given a CSP-CASL specification $SP$ with signature $(\Sigma_{Data}, \Sigma_{Proc})$, its refinement closure $RefCl(SP)$ is defined as follows:

- the signature of $RefCl(SP)$ is that of $SP$,
- the model class of $RefCl(SP)$ consists of those CSP-CASL models $(M', I')$ for which there exists a model $(M, I)$ of $SP$ such that
  - $M = M'$, i.e., they have the same data part,
  - for each $n \in \Sigma_{Proc}$ and all suitable data elements $a_1, \ldots, a_k$,

$$I(n(a_1, \ldots, a_k)) \sqsubseteq I'(n(a_1, \ldots, a_k))$$

  in the sense of CSP.

Alternatively, the semantics of $RefCl(SP)$ can be expressed as a structured specification

$$SP \textbf{ then } p_1 \sqsubseteq q_1, \ldots, p_n \sqsubseteq q_n \textbf{ hide } p_1, \ldots, p_n \textbf{ with } q_1 \mapsto p_1, \ldots, q_n \mapsto p_n$$

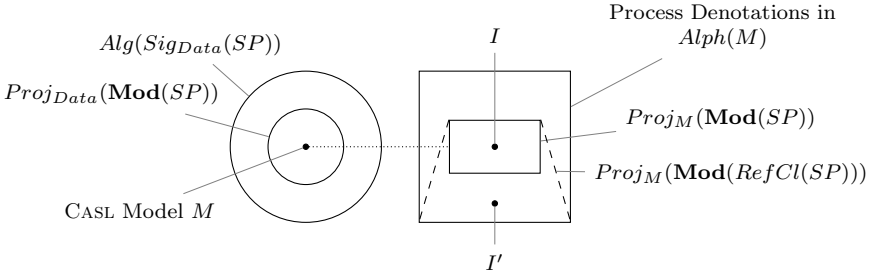



**Fig. 2.** A diagram showing refinement between CSP-CASL models

where $p_1, \ldots, p_n$ are the process names of $SP$ (we assume here that all of them are unparameterised), $q_1, \ldots, q_n$ are new process names, and $p \sqsubseteq q$ stands for $p = p \sqcap q$.

Figure 2 depicts the notion of refinement closure. Given a model $M$ of the data part of $SP$, we consider all of its possible "partners" relative to $SP$: $Proj_M(\mathbf{Mod}(SP)) = \{I \mid (M, I) \in \mathbf{Mod}(SP)\}$ – this is represented by the rectangle. The refinement closure includes all $I'$ such that there exists some $I \in Proj_M(\mathbf{Mod}(SP))$ that refines to $I'$.

With this notion, we are ready to define a notion of refinement that is suitable for CSP-CASL:

$$SP_1 \rightsquigarrow^{\theta}_{\mathcal{D}} SP_2 \text{ iff } \mathbf{Mod}_{\mathcal{D}}(SP_2)|_\theta \subseteq \mathbf{Mod}_{\mathcal{D}}(RefCl(SP_1))$$

for $\mathcal{D} \in \{\mathcal{T}, \mathcal{N}, \mathcal{F}\}$. We omit $\mathcal{D}$ if it is clear from the context and we also omit $\theta$ if it is the identity signature morphism. We write $\_|_\theta$ to denote the CSP-CASL reduct functor. This notion reconciles CASL refinement based on model class inclusion with CSP refinement based on inclusion of trace sets, failure sets, etc. Two specifications $SP_1$ and $SP_2$ are *equivalent*, written $SP_1 \equiv SP_2$, if their signatures and model classes coincide.

**Proposition 4 (Basic Refinement Properties).**

1. *RefCl is monotonic, that is: if* $\mathbf{Mod}(SP_1) \subseteq \mathbf{Mod}(SP_2)$, *then* $\mathbf{Mod}(RefCl(SP_1)) \subseteq \mathbf{Mod}(RefCl(SP_2))$.
2. *RefCl is idempotent, that is* $RefCl(SP) \equiv RefCl(RefCl(SP))$.
3. $\rightsquigarrow$ *is reflexive and transitive.*
4. *If* $SP_1 \rightsquigarrow SP_2$ *and* $SP_2 \rightsquigarrow SP_1$, *then* $RefCl(SP_1) \equiv RefCl(SP_2)$.
5. *If* $SP_1 \rightsquigarrow SP_2$, $SP_2 \rightsquigarrow SP_1$, *and both are single-valued, then* $SP_1 \equiv SP_2$.

Following ideas given in [11] we obtain a decomposition theorem for basic (or unstructured) specifications. This allows us to (syntactically) decompose a basic CSP-CASL specification[5] $SP$ into a data part ($D$) and a process part ($P$), which we shortly write as $(D, P)$.

[5] Such a specification may have a structured CASL specification as the data part $D$.

**Proposition 5 (Decomposition).**

$$\frac{\mathbf{Mod}(D')|_\sigma \subseteq \mathbf{Mod}(D) \qquad (D', \theta(P)) \rightsquigarrow_{\mathcal{D}} (D', P')}{(D, P) \rightsquigarrow^{\theta}_{\mathcal{D}} (D', P')}$$

*where* $\theta = (\sigma, \nu)$ *is a* CSP-CASL *signature morphism, and* $\mathcal{D} \in \{\mathcal{T}, \mathcal{N}, \mathcal{F}\}$.

The above proposition allows us to decompose a CSP-CASL refinement to a CASL refinement (i.e., $\mathbf{Mod}(D')|_\sigma \subseteq \mathbf{Mod}(D)$) and a process refinement $(D', \theta(P)) \rightsquigarrow_{\mathcal{D}} (D', P')$. Here, $\theta(P)$ is the renaming of the process part along $\theta$. The former proof obligation can be discharged using CASL's proof tool, namely HETS [13]. The latter can be proven using the tool CSP-CASL-Prover [17].

### 4.2 Compositional Proof Rules along Structuring

The results of Section 3 allow us to re-use institution independent structuring operations of CASL [15, 22], which are defined in terms of signatures and models:

*Presentations:* For any CSP-CASL signature $\Sigma_{CC}$ and finite set $\Gamma \subseteq \mathbf{Sen}(\Sigma_{CC})$ of $\Sigma_{CC}$-sentences, the presentation $\langle \Sigma_{CC}, \Gamma \rangle$ is a specification with:
$\mathbf{Sig}(\langle \Sigma_{CC}, \Gamma \rangle) := \Sigma_{CC}$
$\mathbf{Mod}(\langle \Sigma_{CC}, \Gamma \rangle) := \{(M, I) \in \mathbf{Mod}(\Sigma_{CC}) \mid (M, I) \models \Gamma\}$

*Union:* For any CSP-CASL signature $\Sigma_{CC}$ and any $\Sigma_{CC}$-specifications $SP_1$ and $SP_2$, their union $SP_1$ **and** $SP_2$ is the specification with:
$\mathbf{Sig}(SP_1 \textbf{ and } SP_2) := \Sigma_{CC}$
$\mathbf{Mod}(SP_1 \textbf{ and } SP_2) := \mathbf{Mod}(SP_1) \cap \mathbf{Mod}(SP_2)$

*Translation:* For any signature morphism $\theta : \Sigma_{CC} \to \Sigma'_{CC}$ and $\Sigma_{CC}$-specification $SP$, $SP$ **rename** $\theta$ is the specification with:
$\mathbf{Sig}(SP \textbf{ rename } \theta) := \Sigma'_{CC}$
$\mathbf{Mod}(SP \textbf{ rename } \theta) := \{(M', I') \in Mod(\Sigma'_{CC}) \mid (M', I')|_\theta \in \mathbf{Mod}(SP)\}$

*Hiding:* For any signature morphism $\theta : \Sigma_{CC} \to \Sigma'_{CC}$ and $\Sigma'_{CC}$-specification $SP'$, $SP'$ **hide** $\theta$ is the specification with:
$\mathbf{Sig}(SP' \textbf{ hide } \theta) := \Sigma_{CC}$
$\mathbf{Mod}(SP' \textbf{ hide } \theta) := \{(M', I')|_\theta \mid (M', I') \in Mod(SP')\}$

As a first proof of concept, we show that the specification building operators are monotonic w.r.t. the structuring operations, cf. [3]. This requires, in our case, certain side conditions, most prominently for the structured union operation on specifications. Here, the conditions deal with the following non-monotonic situation of CSP-CASL refinement: There exist CSP-CASL specifications $SP_1, SP'_1$ and $SP_2$ with[6]

$$SP_1 \rightsquigarrow SP'_1,\ \mathbf{Mod}(SP_1 \textbf{ and } SP_2) = \emptyset,\ \mathbf{Mod}(SP'_1 \textbf{ and } SP_2) \neq \emptyset.$$

[6] Consider over the traces model $SP_1 = (D, P = a \to Stop)$, $SP_2 = (D, P = Stop)$, and $SP'_1 = (D, P = Stop)$ where $D$ is a consistent CASL specification that declares a constant $a$. Then $SP_1$ **and** $SP_2$ is inconsistent, $SP_1 \rightsquigarrow_{\mathcal{T}} SP'_1$, and $SP'_1$ **and** $SP_2$ has models $(M, I)$ with $I(P) = \{\langle\rangle\}$ and $M \in \mathbf{Mod}(D)$.

**Definition 6.** *Two* Csp-Casl *specifications* $SP_1$ *and* $SP_2$ *over the same signature are* process consistent*, written as proc-consistent*$(SP_1, SP_2)$ *if for all* $M \in (Proj_{Data}(\mathbf{Mod}(SP_1)) \cap Proj_{Data}(\mathbf{Mod}(SP_2)))$*, there exists* $(M, J) \in \mathbf{Mod}(SP_1) \cap \mathbf{Mod}(SP_2)$*.*

**Proposition 7.** *The following proof rules*[7] *are sound over* $\mathcal{T}$*,* $\mathcal{N}$*, and* $\mathcal{F}$*:*

$$\frac{SP_1 \rightsquigarrow SP_1' \quad \textit{proc-consistent}(SP_1, SP_2) \quad \textit{single-valued } (SP_i) \textit{ for } i = 1 \vee i = 2}{(SP_1 \textbf{ and } SP_2) \rightsquigarrow (SP_1' \textbf{ and } SP_2)}$$

$$\frac{SP_1 \rightsquigarrow SP_1' \qquad SP_1 \equiv \mathit{RefCl}(SP_1)}{(SP_1 \textbf{ and } SP_2) \rightsquigarrow (SP_1' \textbf{ and } SP_2)}$$

$$\frac{SP \rightsquigarrow SP' \qquad \theta \textit{ is injective on } \text{process names}}{(SP \textbf{ rename } \theta) \rightsquigarrow (SP' \textbf{ rename } \theta)}$$

$$\frac{SP \rightsquigarrow SP'}{(SP \textbf{ hide } \theta) \rightsquigarrow (SP' \textbf{ hide } \theta)}$$

*where* $\theta : \Sigma_{CC} \to \Sigma'_{CC}$*.*

The rules for **and** involve rather strong preconditions, where we hope that it will be possible to obtain better results in the future.

Renaming and refinement involving the same signature morphism can be exchanged:

**Proposition 8.** *The following implications hold:*

1. $(SP \textbf{ rename } \theta) \rightsquigarrow SP'$ *implies* $SP \rightsquigarrow^{\theta} SP'$*.*
2. *Provided that* $\theta$ *is injective on process names, we also have:* $SP \rightsquigarrow^{\theta} SP'$ *implies* $(SP \textbf{ rename } \theta) \rightsquigarrow SP'$*.*

## 5 Compositional Verification of Deadlock Freedom

Our new version of Csp-Casl extends Csp-Casl as was presented in [11] with loose processes. However, our definitions and semantical constructions coincide for single valued specifications.

The deadlock analysis presented in [11] is practically limited to dealing with a small number of processes in parallel. It involves the construction of a so-called sequential process – which has a size that is exponential in the number of parallel components involved. Here we prove deadlock freedom in a far more elegant way.

For the rest of this section, as usual for deadlock analysis in the context of Csp, we work in the stable failures model $\mathcal{F}$ only. Furthermore we assume all processes and process terms to be divergence free.

[7] Note that $\theta$ being injective on process names can have restrictions on the data part of the signature morphism as data forms part of the identity of process names.

### 5.1 Deadlock Freedom in Structured Specifications

We first define what it means for a process term to be deadlock free in the context of a specification (be it basic or structured). We then present a collection of proof rules for deadlock freedom over the structuring operators.

**Definition 9 (Deadlock freedom).** *Let $SP$ be a* Csp-Casl *specification with signature $\Sigma_{CC}$, $X_G$ and $X_L$ be global and local variable systems respectively over $\Sigma_{CC}$, and let $pt$ be a process term over signature $\Sigma_{CC}$ with variable systems $X_G$ and $X_L$. We say: $pt$ is* deadlock free *in specification $SP$, written as*

$$pt\ isDFin\ SP$$

*if for all models $(M, I) \in \mathbf{Mod}(SP)$, for all variable valuations $\mu_G : X_G \to M_\perp$ and $\mu_L : X_L \to M_\perp$, and for all traces $s \in Alph(M)^*$ it holds that $(s, Alph(M)^\checkmark) \notin failures([\![pt]\!]_{(M,I),\mu_G,\mu_L})$.*

Deadlock freedom is compatible with the structuring operations:

**Proposition 10.** *The following proof rules are sound:*

$$\frac{SP \rightsquigarrow^{\theta}_{\mathcal{F}} SP' \qquad pt\ isDFin\ SP}{\theta(pt)\ isDFin\ SP'} \qquad \frac{pt\ isDFin\ SP_1}{pt\ isDFin\ (SP_1\ \mathbf{and}\ SP_2)}$$

$$\frac{pt\ isDFin\ SP}{\theta(pt)\ isDFin\ (SP\ \mathbf{rename}\ \theta)} \qquad \frac{\theta(pt)\ isDFin\ SP'}{pt\ isDFin\ (SP'\ \mathbf{hide}\ \theta)}$$

*where $\theta : \Sigma_{CC} \to \Sigma'_{CC}$.*

The above proof rules allow one to show deadlock freedom by decomposing structured specifications. However, it may still be a difficult task to prove deadlock freedom for complex systems involving parallel processes. We describe a technique for dealing with this situation in the following section.

### 5.2 Composing Networks

In order to study networks of processes, we lift a definition, originally formulated over Csp in [18], to Csp-Casl. This captures the notion of processes being responsive to one and another, i.e., not causing deadlock to occur.

**Definition 11.** *Assume the setting of Definition 9. Let $P$ and $Q$ be process terms over signature $\Sigma_{CC}$ with variable systems $X_G$ and $X_L$. Let $A_P$ and $A_Q$ be downward and upward closed super sets of the constituent alphabet sort set of the process terms $P$ and $Q$ respectively (i.e., $sorts(P) \subseteq A_P$, $\downarrow A_P = A_P$, and $\uparrow A_P = A_P$, similar for $A_Q$)*[8]*, and let $J = A_P \cap A_Q$ be the set of all shared communications sorts. Let $J' \in J_\downarrow$ and $X = \overline{J'} \cup \{\checkmark\}$. Then we define:*

$$Q :: A_Q\ ResToLive^\checkmark\ P :: A_P\ on\ J'\ in\ SP$$

[8] Upward closure is defined in the obvious way: $\uparrow X = \{y \in S \mid \exists x \in X : x \leq y\}$. The condition "upward and downward closed" is required due to Casl subsorting. It ensures that the sort set $J$ comprises all shared communications.

*if for all models* $(M, I) \in \mathbf{Mod}(SP)$*, all variable valuations* $\mu_G : X_G \to M_\perp$ *and* $\mu_L : X_L \to M_\perp$*, and for all traces* $s \in Alph(M)^\checkmark$ *it holds that*

$$(s, X) \in failures([\![P \,|[\, J \,]|\, Q]\!]_{(M,I),\mu_G,\mu_L}) \Longrightarrow (s, X) \in failures([\![P]\!]_{(M,I),\mu_G,\mu_L})$$

In this definition, $Q$ can be seen as a server and $P$ as a client. The server is responsive to the client if whenever the client needs participation from the server, the server is prepared to engage in it.

A *network* is a special way of defining a process in Csp. Formally, a network $V$ is a finite set of pairs $\{(P_i, A_i) \,|\, i \in G\}$, where $G$ is a nonempty, finite index set, $P_i$ is a Csp process, and $A_i \subseteq A$ is the set of communications which $P_i$ can engage in, for all $i \in G$. The process defined by such a network $V$ is

$$Network(V) := \|_{i \in G}\ (P_i, A_i)$$

where $\|_{i \in G}\ (P_i, A_i)$ is the replicated alphabetised parallel operator of Csp. As the semantics of $\|_{i \in G}$ is independent of the order of its arguments, it is sufficient to define networks over index sets. A network $Network(\{P\})$ over a single process $P$ is equivalent to the process $P$ itself. Note that the process *System* in Figure 1 is defined as the network consisting of the processes *Coordinator, Customer, Warehouse,* and *PaymentSystem* with suitable communication sets. Deadlock freedom of such networks can be proven in a compositional way:

**Proposition 12.** *Given a* Csp-Casl *specification* $SP$ *and process terms* $P_i$ *for* $1 \leq i \leq k$ *and a process term* $Q$*. Let* $A_i$ *and* $A_Q$ *be downward and upwards closed supersets of the constituent alphabet of* $P_i$ *for* $1 \leq i \leq k$ *and* $Q$ *respectively. If*

- $A_i \cap A_j \cap A_Q = \emptyset$ *for all* $i$ *and* $j$ *where* $1 \leq i, j \leq k$ *and* $i \neq j$*,*
- $A_i \cap A_Q \neq \emptyset$ *for at least one* $i$ *where* $1 \leq i \leq k$*,*
- $Network(\{(P_1, A_1), \ldots, (P_k, A_k)\})$ *isDFin* $SP$*, and*
- $Q :: A_Q$ *ResToLive*$^\checkmark$ $P_i :: A_i$ *on* $(A_i \cap A_Q)$ *in* $SP$ *for each* $i$ *where* $1 \leq i \leq k$ *and* $A_i \cap A_Q \neq \emptyset$

*then* $Network(\{(P_1, A_1), \ldots, (P_k, A_k), (Q, A_Q)\})$ *isDFin* $SP$*.*

This proposition provides an elegant proof technique: The network under consideration becomes smaller; the property "responds to live" has a characterisation in terms of refinement and thus can be proven, e.g., by Csp-Casl-Prover; the conditions concerning communication alphabets can be proven algorithmically. In order to lift this proof technique to structured specifications, we provide a proof calculus with regards to the property "responds to live":

**Proposition 13.** *The following proof rules are sound:*

$$\frac{Q :: A_Q \ ResToLive^\checkmark \ P :: A_P \ on \ J' \ in \ SP_1}{Q :: A_Q \ ResToLive^\checkmark \ P :: A_P \ on \ J' \ in \ (SP_1 \ \mathbf{and} \ SP_2)}$$

$$\frac{Q :: A_Q \ ResToLive^\checkmark \ P :: A_P \ on \ J' \ in \ SP}{\theta(Q) :: \sigma(A_Q) \ ResToLive^\checkmark \ \theta(P) :: \sigma(A_P) \ on \ \sigma(J') \ in \ (SP \ \mathbf{rename} \ \theta)}$$

$$\frac{\theta(Q) :: \sigma(A_Q)\ ResToLive^{\checkmark}\ \theta(P) :: \sigma(A_P)\ on\ \sigma(J')\ in\ SP'}{Q :: A_Q\ ResToLive^{\checkmark}\ P :: A_P\ on\ J'\ in\ (SP'\ \mathbf{hide}\ \theta)}$$

*where* $\theta = (\sigma, \nu) : \Sigma_{CC} \to \Sigma'_{CC}$.

The above propositions illustrate the successful application of techniques from Csp together with the institution independent structuring mechanisms. We expect other techniques from Csp also to lift successfully to Csp-Casl.

## 6 Example: Online-Shop

In this section we present a proof typical for our calculus. It concerns an online shopping system as has been studied in the literature several times.

### 6.1 The Specification in Detail

The online shop is a typical distributed system. It has several components, namely a customer, a warehouse, a payment system, and a coordinator. The communication structure is pointwise only: The coordinator communicates with the three other components in a star like network. The customer, warehouse and payment system only communicate with the coordinator.

The customer may ask the coordinator to perform actions such as: To login, to add an item to the basket, to remove an item from the basket, to checkout, etc. The coordinator then responds to the customer with an appropriate response message. All communication (on a channel) follows this pattern of a request message followed by a response message (except for the *Logout* message, which is more of a command). The coordinator may ask the warehouse to reserve an item, to release an item that has previously been reserved, and to dispatch the reserved items. The payment system allows the coordinator to take payments for goods.

We specify the shop on various levels of abstraction. The architectural shop (see Figure 1) describes the network layout, which remains unchanged in the development. The development is restricted to individually refining the four components. Here we present the first two levels of abstraction for our example, namely: The architectural level for describing the basic interfaces, and the abstract component level (ACL) for specifying the type system and its interplay with process behaviour. Each component contains a 'main' process as its starting point.

Within the rest of this section we use $C$ to denote customer, $Co$ to denote coordinator, $W$ to denote warehouse and $PS$ to denote payment system. We also drop the communications sets within the network construction and take them to be the declared communications sets of the process names implicitly, for instance: by $Network(\{Customer\})$, we mean $Network(\{(Customer, C\_C)\})$.

**spec** ACL_Customer =
**data sorts** *LoginReq*, *Logout*, *GoodLoginRes*, *BadLoginRes*, *AddItemReq*, *AddItemRes*, ... < *D_C*
**channel** *C_C* : *D_C*
**process** *Customer* : *C_C* ; *Customer_GoodLogin* : *C_C* ;
*Customer_BadLogin* : *C_C* ; *Customer_AddItem* : *C_C* ;
*Customer_Body* : *C_C* ; *Customer_Quit* : *C_C* ; ...
*Customer* = *C_C* ! *x* :: *LoginReq_def* →
(*Customer_GoodLogin* ; *Customer_Body* □
*Customer_BadLogin* ; *Customer*)
*Customer_GoodLogin* = *C_C* ? *x* :: *GoodLoginRes_def* → *SKIP*
*Customer_BadLogin* = *C_C* ? *x* :: *BadLoginRes_def* → *SKIP*
*Customer_AddItem* = *C_C* ! *x* :: *AddItemReq_def* →
*C_C* ? *y* :: *AddItemRes_def* →
*Customer_Body*
...
*Customer_Quit* = *C_C* ! *x* :: *Logout_def* → *SKIP*
*Customer_Body* = *Customer_AddItem* ⊓ ...
⊓ *Customer_Quit*
**end**

**spec** ACL_Coordinator =
**data** ...
**channels** *C_C* : *D_C*; *C_W* : *D_W*; *C_PS* : *D_PS*
**process** ...
*Coordinator_AddItem* = *C_C* ? *x* :: *AddItemReq_def* →
*C_W* ! *y2* :: *ReserveItemReq_def* →
*C_W* ? *x2* :: *ReserveItemRes_def* →
*C_C* ! *y* :: *AddItemRes_def* →
*Customer_Body*
*Coordinator_Body* = *Coordinator_AddItem* □ ...
□ *Coordinator_Quit*
**end**

**Fig. 3.** Specification of the ACL customer and coordinator specifications

### 6.2 Deadlock Analysis

We illustrate how to prove deadlock freedom using the technique presented in Section 5. We discuss the core part of the proof, and explain how to scale it up for the whole system. The proof rule from Proposition 12 reduces the network of processes step by step. We start at the point where the network has been reduced to two processes only:

**spec** Reduced_Arch_Shop [*RefCl*(Arch_C)] [*RefCl*(Arch_Co)] =
**process** *System′* : *C_C* ;
*System′* = *Coordinator* ⟦ *C_C* || *C_C* ⟧ *Customer*
**end**

The specification REDUCED_ARCH_SHOP instantiated with ACL components is semantically equivalent to the following specification (without parameterisation):

REDUCED_ACL_SHOP =
(( $(ReflCl($ARCH_C$)$ **rename** $\theta_1$) **and**
$(ReflCl($ARCH_CO$)$ **rename** $\theta_2$) **and** BODY
) **rename** $\theta_3$) **and** (ACL_C **rename** $\theta_4$) **and** (ACL_CO **rename** $\theta_5$)

Here all signature morphisms involved are embeddings and the specification BODY is a basic specification with the signature equal to the union of the signatures of the ACL customer and coordinator along with the new process name $System'$, and where the only axiom is that of

$System' = Coordinator \; [\![ \; C\_C \; || \; C\_C \; ]\!] \; Customer.$

Our aim is to prove that the process term bound to $System'$ is deadlock free within the specification REDUCED_ACL_SHOP. To this end, we apply Proposition 12 and obtain:

$Network(\{Customer, Coordinator\})$ $isDFin$ REDUCED_ACL_SHOP
if (a) $C$ $isDFin$ REDUCED_ACL_SHOP and
(b) $Co :: C\_C$ $ResToLive^{\checkmark}$ $C :: C\_C$ $on$ $C\_C$ $in$ REDUCED_ACL_SHOP

To discharge obligation (a), we apply the **and** rule from Proposition 10 several times and reduce it to ($C$ $isDFin$ ACL_C **rename** $\theta_4$). Applying the renaming rule (also from Proposition 10) results in ($C$ $isDFin$ ACL_C). As ACL_C is a basic specification and the customer process does not involve any parallel operator we can easily discharge this obligation with CSP-CASL-Prover.

Concerning obligation (b), we apply the **and** rule from Proposition 13 several times and reduce it to:

$Co :: C\_C$ $ResToLive^{\checkmark}$ $C :: C\_C$ $on$ $C\_C$ $in$
((ACL_C **rename** $\theta_4$) **and** (ACL_CO **rename** $\theta_5$))

As ACL_C and ACL_CO are basic CSP-CASL specifications we can discharge the proof obligation by applying the flattening operation and then using CSP-CASL-Prover. This obligation holds because the coordinator allows the customer to choose the initial action (a request message) and then provides a response message to the customer for this particular type of request (possibly after further communications with other components).

The full proof of deadlock freedom has the same structure. Proposition 12 reduces $Network(\{Customer, Coordinator, PaymentSystem, Warehouse\})$ down to $Network(\{Customer\})$ by removing first $Warehouse$, then $PaymentSystem$, and – as shown above – $Customer$ from the network. The resulting obligations can then be reduced to a format where they can be discharged with CSP-CASL-Prover.

## 7 Conclusion and Future Work

We have presented institutions for CSP-CASL, where we added the new feature of loose process semantics. This setting allowed us to define and study structuring,

parameterisation and refinement of Csp-Casl specifications. We gave several proof calculi for compositional reasoning along the structure of Csp-Casl specifications: One dedicated to refinement, the other for deadlock analysis.

Future work will include the development of further proof rules for structured operations: We intend to improve the refinement rules for **and**, and we want to develop proof rules for the structured **free** operation, with a special emphasis on connection with the Csp fixed point theory. Furthermore, we plan to apply structuring to our EP2 case study, and to implement the presented calculi within the standard proof tool for Casl, namely, Hets [13].

**Acknowledgement.** The authors are grateful to Erwin Catesbeiana for his structured advice on how to navigate through deadlocked situations. This work has been supported by the German Federal Ministry of Education and Research (Project 01 IW 10002 SHIP).

## References

[1] eft/pos 2000 Specification, version 1.0.1. EP2 Consortium (2002)
[2] Allen, R., Garlan, D.: A formal basis for architectural connection. ACM Trans. Softw. Eng. Methodol. 6(3), 213–249 (1997)
[3] Bidoit, M., Cengarle, V.V., Hennicker, R.: Proof systems for structured specifications and their refinements. In: Astesiano, E., Kreowski, H.-J., Krieg-Brückner, B. (eds.) Algebraic Fondations of System Specification, pp. 385–434. Springer, Heidelberg (1999)
[4] Bidoit, M., Mosses, P.D. (eds.): CASL User Manual. LNCS, vol. 2900. Springer, Heidelberg (2004)
[5] Cerioli, M., Meseguer, J.: May I borrow your logic (Transporting logical structures along maps). Theoretical Computer Science 173, 311–347 (1997)
[6] Diaconescu, R., Goguen, J., Stefaneas, P.: Logical support for modularisation. In: Logical Environments, Cambridge, pp. 83–130 (1993)
[7] Fischer, C.: How to Combine Z with a Process Algebra. In: P. Bowen, J., Fett, A., Hinchey, M.G. (eds.) ZUM 1998. LNCS, vol. 1493, pp. 5–25. Springer, Heidelberg (1998)
[8] Gimblett, A., Roggenbach, M., Schlingloff, B.-H.: Towards a Formal Specification of an Electronic Payment System in CSP-CASL. In: Fiadeiro, J.L., Mosses, P.D., Yu, Y. (eds.) WADT 2004. LNCS, vol. 3423, pp. 61–78. Springer, Heidelberg (2005)
[9] Goguen, J.A., Burstall, R.M.: Institutions: Abstract model theory for specification and programming. J. ACM 39(1), 95–146 (1992)
[10] Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall (1985)
[11] Kahsai, T., Roggenbach, M.: Property Preserving Refinement for Csp-Casl. In: Corradini, A., Montanari, U. (eds.) WADT 2008. LNCS, vol. 5486, pp. 206–220. Springer, Heidelberg (2009)
[12] Mossakowski, T.: ModalCASL. Language Summary (2004), `http://www.informatik.uni-bremen.de/~till/papers/Modal-Summary.pdf`
[13] Mossakowski, T., Maeder, C., Lüttich, K.: The Heterogeneous Tool Set, Hets. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 519–522. Springer, Heidelberg (2007)

[14] Mossakowski, T., Roggenbach, M.: Structured CSP – A Process Algebra as an Institution. In: Fiadeiro, J.L., Schobbens, P.-Y. (eds.) WADT 2006. LNCS, vol. 4409, pp. 92–110. Springer, Heidelberg (2007)
[15] Mosses, P.D. (ed.): CASL Reference Manual. LNCS, vol. 2960. Springer, Heidelberg (2004)
[16] O'Reilly, L., Kahsai, T., Mossakowski, T., Roggenbach, M.: The CSP-CASL institution. Technical Report CSR-1-2011, Swansea University (2011)
[17] O'Reilly, L., Roggenbach, M., Isobe, Y.: CSP-CASL-Prover: A generic tool for process and data refinement. ENTCS 250(2), 69–84 (2009)
[18] Reed, J.N., Sinclair, J.E., Roscoe, A.W.: Responsiveness of interoperating components. Formal Asp. Comput. 16(4), 394–411 (2004)
[19] Reggio, G., Astesiano, E., Choppy, C.: CASL-LTL. Technical Report DISI-TR-99-34, Università di Genova (2000)
[20] Roggenbach, M.: CSP-CASL: A new integration of process algebra and algebraic specification. Theoretical Computer Science 354(1), 42–71 (2006)
[21] Roscoe, A.W.: Understanding Concurrent Systems. Springer, Heidelberg (2010)
[22] Sannella, D., Tarlecki, A.: Specifications in an arbitrary institution. Information and Computation 76, 165–210 (1988)
[23] Scattergood, B.: The semantics and implementation of machine-readable CSP. PhD thesis, Oxford University (1998)
[24] Wehrheim, H.: Behavioural subtyping in object-oriented specification formalisms, Habilitation thesis, Carl-von-Ossietzky-Universität Oldenburg (2002)
[25] Zawłocki, A.: Architectural Specifications for Reactive Systems. In: Fiadeiro, J.L., Mosses, P.D., Yu, Y. (eds.) WADT 2004. LNCS, vol. 3423, pp. 252–269. Springer, Heidelberg (2005)