Property preserving refinement for CSP-CASL

Temesghen Kahsai, Markus Roggenbach [*]

Department of Computer Science, Swansea University, UK.
{csteme, csmarkus}@swan.ac.uk

Abstract. In this paper we present various notions of the combined refinement for data and processes within the specification language CSP-CASL. We develop proof support for our refinement notions and demonstrate how to employ them for system development and for system analysis. Finally, we apply our technique to an industrial standard for an electronic payment system.

1 Introduction

System development in a step-by-step fashion has been central to software engineering at least since Wirth's seminal paper on program development [22] in 1971. Such a development starts with an abstract specification, which defines the general setting, e.g. it might define the components and interfaces involved in the system. In several design steps this abstract specification is then further developed towards a design specification which can be implemented directly. In each of these steps some design decisions are taken and implementation issues are resolved. A design step can for instance refine the type system, or it might set up a basic dialogue structure. It is essential, however, that these design steps preserve properties. This idea is captured by the notion of *refinement*.

In industrial practice, stepwise development usually is carried out informally. In this paper, we capture such informal developments with formal notions of refinement within the specification language CSP-CASL [19]. CSP-CASL allows one to specify data and processes in an integrated way, where CASL [17] is used to describe data and CSP [7,20] is used to specify the reactive side.

Our notions of refinement for CSP-CASL are based on refinements developed in the context of the single languages CSP and CASL. In the context of algebraic specification, e.g., [4] provides an excellent survey on different approaches. For CSP, each of its semantical models comes with a refinement notion of its own. There are for instance traces refinement, failure/divergences refinement, and stable failures refinement [20]. For *system development* one often is interested in liberal notions of refinements, which allow substantial changes in the design. For *system verification*, however, it is important that refinement steps preserve properties. The latter concept allows one to verify properties already on abstract specifications – which in general are less complex than the more concrete ones. The properties, however, are preserved over the design steps. These two purposes motivate our study of various refinement notions.

In this paper, we develop proof methods for CSP-CASL. To this end, we decompose a CSP-CASL refinement into a refinement over CSP and a refinement over CASL alone.

[*] This work was supported by EPSRC under the grant EP/D037212/1.

We show how to use existing tools to discharge the arising proof obligations. Reactive systems often exhibit the undesirable behaviour of deadlock or divergence (livelock), which both result in lack of progress in the system. Here, we develop proof techniques based on refinement for proving deadlock freeness and divergence freeness.

The language CSP dates back at least to 1985 [7]; an excellent reference is the book [20] (updated 2005). CASL was designed by the CoFI initiative [17]. Tools for CASL are developed, e.g., in [11,12,13]. The combination CSP-CASL was introduced in [19] and used for specifying an electronic payment system in [6]. A tool for CSP refinement was developed in [9]. [5] implements a parser for CSP-CASL, [18] describes a theorem prover for CSP-CASL. In [16], Mossakowski et al. define a refinement language for CASL architectural specifications. In [1] Woodcock et al. discuss a proof-by-refinement technique in the area of *Z* specification. Deadlock analysis in CSP has been studied in [20], and an industrial application has been described in [3]. Tools for deadlock analysis are developed in [10,8]. Livelock analysis in CSP has been applied to an industrial application in [21].

In the next section we give an overview of the specification language CSP-CASL and refinement based on model class inclusion. Section 3 we develop proof support for CSP-CASL refinement and describe how refinement can be employed for deadlock and divergence analysis In CSP-CASL. In Section 4 we demonstrate that the presented theoretical results are applicable in an industrial setting.

2 CSP-CASL

CSP-CASL [19] is a specification language which combines *processes* written in CSP [7,20] with the specification of *data types* in CASL [17]. 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, 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 sub-sorting. Furthermore, the various CASL structuring constructs are included, where the structured **free** construct adds the possibility to specify data types with initial semantics.

Syntactically, a CSP-CASL specification with a name *Sp* consists of a data part *D*, which is a structured CASL specification, an (optional) channel part *Ch* to declare channels, which are typed according to the data part, and a process part *P* written in CSP, within which CASL terms are used as communications, CASL sorts denote sets of communications, relational renaming is described by a binary CASL predicate, and the CSP conditional construct uses CASL formulae as conditions:

ccspec *Sp* = **data** *D* **channel** *Ch* **process** *P* **end**

For concrete examples of CSP-CASL specifications see Section 4. The CSP-CASL channel part is syntactic sugar over the data part, see [19] for the details of the encoding into CASL. In our practical examples we will make use of channels. For our semantical considerations, however, we will study specifications only without channels. We often write such specifications shortly as $Sp = (D,P)$.

Semantically, a CSP-CASL specification $Sp = (D,P)$ is a family of process denotations for a CSP process, where each model of the data part D gives rise to one process denotation. CSP-CASL has a 2-steps semantics. In the first step we construct for each CASL model $M \in \mathbf{Mod}(D)$ a CSP process $[\![P]\!]_M$, which communicates in an alphabet $\mathcal{A}(M)$ constructed out of the CASL model M. In the second step we point-wise apply a denotational CSP semantics. This translates a process $[\![P]\!]_M$ into its denotation d_M in the semantic domain of the chosen CSP model. The overall semantical construction is written $([\![[\![P]\!]_M]\!]_{\mathrm{CSP}})_{M\in\mathbf{Mod}(D)}$. For a denotational CSP model with domain $\mathcal{D}$, the semantic domain of CSP-CASL consists of families of process denotations $d_M \in \mathcal{D}$ over some index set I, $(d_M)_{M\in I}$, where I is a class of CASL models over the same signature.

CSP-CASL refinement is based on refinements developed in the context of the single languages CSP and CASL. Intuitively, a refinement step, which we write here as '$\rightsquigarrow$', reduces the number of possible implementations. Concerning data, this means a reduced model class, concerning processes this mean less non-deterministic choice:

Definition 1 (Model class inclusion). *For families $(d_M)_{M\in I}$ and $(d'_{M'})_{M'\in I'}$ of process denotations we write $(d_M)_{M\in I} \rightsquigarrow_{\mathcal{D}} (d'_{M'})_{M'\in I'}$ iff $I' \subseteq I \wedge \forall M' \in I' : d_{M'} \sqsubseteq_{\mathcal{D}} d'_{M'}$.*

Here, $I' \subseteq I$ denotes inclusion of model classes over the same signature, and $\sqsubseteq_{\mathcal{D}}$ is the refinement notion in the chosen CSP model $\mathcal{D}$. In the traces model $\mathcal{T}$ we have for instance $P \sqsubseteq_{\mathcal{T}} P' \Leftrightarrow traces(P') \subseteq traces(P)$, where $traces(P)$ and $traces(P')$ are prefixed closed sets of traces. Here we follow the CSP convention, where P' *refines* P is written as $P \sqsubseteq_{\mathcal{D}} P'$, i.e. the more specific process is on the right-hand side of the symbol. The definitions of CSP refinements for $\mathcal{D} \in \{\mathcal{T},\mathcal{N},\mathcal{F},\mathcal{I},\mathcal{U}\}$, c.f. [20], which are all based on set inclusion, yield that CSP-CASL refinement is a preorder.

Given CSP-CASL specifications $Sp = (D,P)$ and $Sp' = (D',P')$, by abuse of notation we also write $(D,P) \rightsquigarrow_{\mathcal{D}} (D',P')$ if the above refinement notion holds for the denotations of Sp and Sp', respectively.

On the syntactic level of specification text, we additionally define the notions of data refinement and process refinement in order to characterize situations, where one specification part remains constant. In a *data refinement*, only the data part changes:

$$\left.\begin{array}{c}\textbf{data } D \textbf{ process } P \textbf{ end}\\ \overset{\mathtt{data}}{\rightsquigarrow}\\ \textbf{data } D' \textbf{ process } P \textbf{ end}\end{array}\right\} \quad \text{if} \quad \left\{\begin{array}{l}1.\ \Sigma(D) = \Sigma(D'),\\ 2.\ \mathbf{Mod}(D') \subseteq \mathbf{Mod}(D)\end{array}\right.$$

Here, $\Sigma(D)$ denotes the CASL signature of D. As in a data refinement the process part remains the same, there is no need to annotate data refinement with a specific process model: all CSP refinements notions are reflexive. In a *process refinement*, the data part is constant:

$$\left.\begin{array}{c}\textbf{data } D \textbf{ process } P \textbf{ end}\\ \overset{\mathtt{proc}}{\rightsquigarrow}_{\mathcal{D}}\\ \textbf{data } D \textbf{ process } P' \textbf{ end}\end{array}\right\} \quad \text{if} \quad \left\{\begin{array}{l}\text{for all } M \in \mathbf{Mod}(D) :\\ [\![[\![P]\!]_M]\!]_{\mathrm{CSP}} \sqsubseteq_{\mathcal{D}} [\![[\![P']\!]_M]\!]_{\mathrm{CSP}}\end{array}\right.$$

Clearly, both these refinements are special forms of CSP-CASL refinement in general.

3 A basic theory of CSP-CASL refinement

In this section we develop proof support for CSP-CASL refinement, and theories of how to analyse CSP-CASL specifications.

3.1 Proof Support

Proof support for the CSP-CASL refinement is based on a decomposition theorem. This decomposition theorem gives rise to a proof method for CSP-CASL, namely, we study CSP-CASL refinement in terms of CASL refinement and CSP refinement separately. With regard to CSP-CASL refinement, data turns out to dominate the processes: While any CSP-CASL refinement can be decomposed into first a data refinement followed by a process refinement, there is no such decomposition result possible for the reverse order. This insight is in accordance with the 2-step semantics of CSP-CASL, where in the first step we evaluate the data part and only in the second step apply the process semantics.

First, we present our positive result concerning decomposition:

Theorem 1. *Let $Sp = (D,P)$ and $Sp' = (D',P')$ be* CSP-CASL *specifications, where D and D' are data specifications over the same signature. Let (D',P) be a new* CSP-CASL *specification. For these three specifications holds: $(D,P) \rightsquigarrow_{\mathcal{D}} (D',P')$ iff $(D,P) \overset{data}{\rightsquigarrow} (D',P)$ and $(D',P) \overset{proc}{\rightsquigarrow}_{\mathcal{D}} (D',P')$.*

This result forms the basis for the CSP-CASL tool support developed in [18]. In order to prove that a CSP-CASL refinement $(D,P) \rightsquigarrow_{\mathcal{D}} (D',P')$ holds, first one uses proof support for CASL [13] alone in order to establish $\mathbf{Mod}(D') \subseteq \mathbf{Mod}(D)$. Independently of this, one has then to check the process refinement $P \sqsubseteq_{\mathcal{D}} P'$. In principle, the latter step can be carried out using CSP-Prover, see e.g. [9]. The use of CSP-Prover, however, requires the CASL specification D' to be translated into an alphabet of communications. The tool CSP-CASL-Prover [18] implements this translation and also generates proof support for theorem proving on CSP-CASL.

Changing the order in the above decomposition theorem, i.e., to first perform first a process refinement followed by a data refinement, however, is not possible in general. Often, process properties depend on data, as the following counter example illustrates, in which we have: $(D,P) \rightsquigarrow_{\mathcal{T}} (D',P')$ but $(D,P) \not\rightsquigarrow_{\mathcal{T}} (D,P')$. Consider the three CSP-CASL specifications ABS, MID and CONC, where MID consists of the data part of ABS and the process part of CONC:

ccspec ABS =
  data
    sorts S
    ops $a, b : S;$
  process
    P $= a \rightarrow Stop$
end

ccspec MID =
  data
    sort S
    ops $a, b : S;$
  process
  Q $= a \rightarrow Stop \,|[\, a \,]|$
    $b \rightarrow Stop$
end

ccspec CONC =
  data
    sort S
    ops $a, b : S;$
    axiom $a = b$
  process
  R $= a \rightarrow Stop \,|[\, a \,]|$
    $b \rightarrow Stop$
end

Let N be a CASL model of the data part D_{ABS} of ABS with $N(S) = \{\#, *\}$, $N(a) = \#$, $N(b) = *$. Concerning the process denotations in the traces model $\mathcal{T}$ relatively to N, for ABS we obtain the denotation[1] $d_{\text{ABS}} = \{\langle\rangle, \langle\#\rangle\}$. In MID, the alphabetized parallel operator requires synchronization only w.r.t. the event a. As $N \models \neg a = b$, the rhs of the parallel operator, which is prepared to engage in b, can proceed with b, which yields the trace $\langle * \rangle$ in the denotation. The lhs, however, which is prepared to engage in a, does not find a partner for synchronization and therefore is blocked. This results in the denotation $d_{\text{MID}} = \{\langle\rangle, \langle * \rangle\}$. As $d_{\text{MID}} \nsubseteq d_{\text{ABS}}$, we have ABS $\not\rightsquigarrow_{\mathcal{T}}$ MID.

In CONC, the axiom $a = b$ prevents N to be a model of the data part. This makes it possible to establish ABS $\rightsquigarrow_{\mathcal{T}}$ CONC over the traces model $\mathcal{T}$. Using Theorem 1, we first prove the data refinement: CONC adds an axiom to ABS – therefore, D_{ABS} refines to D_{CONC} with respect to CASL; concerning the process refinement, using the equation $a = b$ and the step law for generalized parallel, we obtain $a \to \mathit{Stop} \,|[\, a \,]|\, b \to \mathit{Stop} =_{\mathcal{T}} a \to \mathit{Stop} \,|[\, a \,]|\, a \to \mathit{Stop} =_{\mathcal{T}} a \to (\mathit{Stop} \,|[\, a \,]|\, \mathit{Stop}) =_{\mathcal{T}} a \to \mathit{Stop}$ – thus, over D_{CONC} the process parts of ABS CONC are semantically equivalent and therefore in refinement relation over the traces model $\mathcal{T}$.

3.2 Analysis for Deadlock Freeness

In this section we show how to analyse deadlock freeness in the context of CSP-CASL. To this end, first we recall how deadlock is characterized in CSP. Then we define what it means for a CSP-CASL specification to be deadlock free. Finally, we establish a proof technique for deadlock freeness based on CSP-CASL refinement, which turns out to be complete.

In the CSP context, the stable failures model $\mathcal{F}$ is best suited for deadlock analysis. The stable failures model $\mathcal{F}$ records two observations on processes: the first observation is the set of traces a process can perform, this observation is given by the semantic function *traces*; the second observation are the so-called stable failures, given by the semantic function *failures*. A failure is a pair (s, X), where s represents a trace that the process can perform, after which the process can refuse to engage in all events of the set X. We often write (T, F) for such a pair of observations, T denoting the set of traces and F denoting the set of stable failures. Deadlock is represented by the process *STOP*. Let A be the alphabet. Then the process *STOP* has

$$(\{\langle\rangle\}, \{(\langle\rangle, X) \mid X \subseteq A^{\checkmark}\}) \in \mathcal{P}(A^{*\checkmark}) \times \mathcal{P}(A^{*\checkmark} \times \mathcal{P}(A^{\checkmark}))$$

as its denotation in $\mathcal{F}$, i.e., the process *STOP* can perform only the empty trace, and after the empty trace the process *STOP* can refuse to engage in all events. Here, $\checkmark \notin A$ is a special event denoting successful termination, $A^{\checkmark} = A \cup \{\checkmark\}$, and $A^{*\checkmark} = A^* \cup A^{*\frown}\langle\checkmark\rangle$ is the set of all traces over A possibly ending with $\checkmark$. In CSP, a process P is considered to be deadlock free, if the process P after performing a trace s never becomes equivalent to the process *STOP*. More formally: A process P is **deadlock-free** in CSP iff

$$\forall s \in A^*.(s, A^{\checkmark}) \notin \mathit{failures}(P).$$

[1] For the sake of readability, we write the element of the carrier sets rather than their corresponding events in the alphabet of communications.

This definition is justified, as in the model $\mathcal{F}$ the set of stable failures is required to be closed under subset-relation: $(s,X) \in \mathit{failures}(P) \wedge Y \subseteq X \Rightarrow (s,Y) \in \mathit{failures}(P)$. In other words: Before termination, the process P can never refuse all events; there is always some event that P can perform.

A CSP-CASL specification has a family of process denotations as its semantics. Each of these denotations represents a possible implementation. We consider a CSP-CASL specification to be deadlock free, if it enforces all its possible implementations to have this property. On the semantical level, we capture this idea as follows:

Definition 2. *Let $(d_M)_{M\in I}$ be a family of process denotations over the stable failures model, i.e., $d_M = (T_M, F_M) \in \mathcal{F}(\mathcal{A}(M))$ for all $M \in I$.*

- *d_M is deadlock free if $(s,X) \in F_M$ and $s \in \mathcal{A}(M)^*$ implies that $X \neq \mathcal{A}(M)^{\checkmark}$.*
- *$(d_M)_{M\in I}$ is deadlock free if for all $M \in I$ it holds that d_M is deadlock-free.*

Deadlock can be analyzed trough refinement checking; that is an implementation is deadlock-free if it is the refinement of a deadlock free specification:

Theorem 2. *Let $(d_M)_{M\in I} \rightsquigarrow_{\mathcal{F}} (d'_{M'})_{M'\in I'}$ be a refinement over $\mathcal{F}$ between two families of process denotations. If $(d_M)_{M\in I}$ is deadlock-free, then so is $(d'_{M'})_{M'\in I'}$.*

Following an idea from the CSP context, we formulate the most abstract deadlock free CSP-CASL specification over a subsorted CASL signature $\Sigma = (S, TF, PF, P, \leq)$ – see [17] for the details – with a set of sort symbols $S = \{s_1, \ldots, s_n\}, n \geq 1$:

ccspec DF$_\Sigma$ =
  data ... declaration of Σ ...
  process $DF_S = \bigsqcap_{s:S} (!x : s \rightarrow DF_S) \sqcap Skip$
end

Here, the process $!x : s \rightarrow DF_S$ internally chooses an element x from the sort s, engages in it, and then behaves like DF_S. We observe:

Lemma 1. DF$_\Sigma$ *is deadlock free.*

Proof. Let $(d_M)_{M\in I}$ be the denotation of DF$_\Sigma$ over the stable-failures model, where $d_M = (T_M, F_M)$. For all $M \in I$ holds: $T_M = \mathcal{A}(M)^{*\checkmark}$ and $F_M = \{(t,X) \mid t \in \mathcal{A}(M)^*, X \subseteq \mathcal{A}(M) \vee \exists a \in \mathcal{A}(M).\ X \subseteq \mathcal{A}(M)^{\checkmark} - \{a\}\} \cup \{(t \frown \langle\checkmark\rangle, Y) \mid t \in \mathcal{A}(M)^*, Y \subseteq \mathcal{A}(M)^{\checkmark}\}$.

This result on DF$_\Sigma$ extends to a complete proof method for deadlock freeness in CSP-CASL:

Theorem 3. *A* CSP-CASL *specification (D,P) is deadlock free iff* DF$_\Sigma \rightsquigarrow_{\mathcal{F}} (D,P)$. *Here, Σ is the signature of D.*

Proof. If DF$_\Sigma \rightsquigarrow_{\mathcal{F}} (D,P)$, Lemma 1 and Theorem 2 imply that (D,P) is deadlock free. Now let (D,P) be deadlock free. We apply Theorem 1 to our proof goal DF$_\Sigma \rightsquigarrow_{\mathcal{F}}$ (D,P) and decompose it into a data refinement and a process refinement. The data refinement holds, as the model class of DF$_\Sigma$ consists of all CASL models over Σ. The process refinement holds thanks to the semantics of DF$_\Sigma$ as given in the proof of Lemma 1.

3.3 Analysis for Divergence Freeness

For concurrent systems, divergence (or livelock) is regarded as an individual starvation, i.e., a particular process is prevented from engaging in any actions. For CSP, the failures/divergences model $\mathcal{N}$ is considered best to study systems with regard to divegence. The CSP process **div** represents this phenomenon: immediately, it can refuse every event, and it diverges after any trace. **div** is the least refined process in the $\sqsubseteq_{\mathcal{N}}$ model. The main sources for divergence in CSP are *hiding* and ill-formed recursive processes.

In the failures/divergences model $\mathcal{N}$, a process is modeled as a pair (F, D). Here, F represents the *failures*, while D collects all *divergences*. Let A be the alphabet. The process **div** has

$$(A^{*\checkmark} \times \mathcal{P}(A^{\checkmark}), A^{*\checkmark}) \in \mathcal{P}(A^{*\checkmark} \times \mathcal{P}(A^{\checkmark})) \times \mathcal{P}(A^{*\checkmark})$$

as its semantics over $\mathcal{N}$.

Following these ideas, we define what it means for a CSP-CASL specification to be divergence free: Essentially, after carrying out a sequence of events, the denotation shall be different from **div**.

Definition 3. *Let $(d_M)_{M\in I}$ be a family of process denotations over the failure divergence model, i.e, $d_M = (F_M, D_M) \in \mathcal{N}(\mathcal{A}(M))$ for all $M \in I$.*

- *A denotation d_M is* **divergence free** *iff one of the following conditions holds:*
  C1. $\forall s \in \mathcal{A}(M)^*.\{(t, X) \mid (s ^\frown t, X) \in F_M\} \neq \mathcal{A}(M)^{*\checkmark} \times \mathcal{P}(\mathcal{A}(M)^{\checkmark})$
  C2. $\forall s \in \mathcal{A}(M)^*.\{t \mid (s ^\frown t) \in D\} \neq \mathcal{A}(M)^{*\checkmark}$.
- *$(d_M)_{M\in I}$ is* **divergence free** *if for all $M \in I$ it holds that d_M is divergence free.*

Like in the case of analysis for deadlock freeness, also the analysis for divergence freeness can be checked trough refinement, this time over the model $\mathcal{N}$.

Theorem 4. *Let $(d_M)_{M\in I} \rightsquigarrow_{\mathcal{N}} (d'_{M'})_{M'\in I'}$ be a refinement over $\mathcal{N}$ between two families of process denotations. Let $(d_M)_{M\in I}$ be divergence free. Then $(d'_{M'})_{M'\in I'}$ is divergence free.*

As for the analysis of deadlock freeness we formulate the least refined divergence free CSP-CASL specification over a CASL signature Σ with a set of sort of symbols $S = \{s_1, \dots, s_n\}, n \geq 1$.

ccspec DIVF$_\Sigma$ =
  data ... declaration of Σ ...
  process $DivF = (Stop \sqcap Skip) \sqcap (\Box_{s:S} ?x : s \rightarrow DivF)$
end

DivF may deadlock at any time, it may terminate successfully at any time, or it may perform any event at any time, however, it will not diverge. Figure 1 shows DIVF$_\Sigma$ as a labelled transition system. One can easily see that this transition system does not have a path of infinite silent actions τ. Here, $[s_i]$ denotes the collection of events constructed over the sort s_i. This observation is reflected in the following lemma:

Lemma 2. DIVF$_\Sigma$ *is divergence free.*

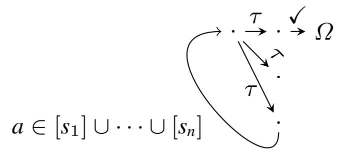

Fig. 1. An LTS version of DIVF$_\Sigma$.

Proof. Let $(d_M)_{M\in I}$ be the semantics of DIVF$_S$ over the failures/divergences model $\mathcal{N}$ where $d_M = (F_M, D_M) \in \mathcal{N}(\mathcal{A}(M))$. For all models M holds: $F_M = \mathcal{A}(M)^{*\checkmark} \times \mathcal{P}(\mathcal{A}(M)^{\checkmark})$ and $D_M = \emptyset$. Thus, DIVF$_\Sigma$ is divergence free thanks to conditions **C.2**.

Putting things together, we obtain a complete proof method for divergence freedom of CSP-CASL specifications:

Theorem 5. *A* CSP-CASL *specification* (D,P) *is divergence free iff* DIVF$_\Sigma \rightsquigarrow_{\mathcal{F}} (D,P)$. *Here* Σ *is the signature of D.*

Proof. If DIVF$_\Sigma \rightsquigarrow_{\mathcal{N}} (D,P)$, Lemma 2 and Theorem 4 imply that (D,P) is divergence free. Now let (D,P) be divergence free. Assume that $\neg($DIVF$_\Sigma \rightsquigarrow_{\mathcal{N}} (D,P))$. As the data part of DIVF refines to D, with our decomposition theorem 1 we can conclude that $\neg((D, DivF) \overset{\texttt{proc}}{\rightsquigarrow}_{\mathcal{N}} (D,P))$. Let $(d_M)_{M\in\mathbf{Mod}(D)}$ be the semantics of $(D, DivF)$, where $d_M = (F_M, D_M)$. Let $(d'_M)_{M\in\mathbf{Mod}(D)}$ be the semantics of (D,P), where $d'_M = (F'_M, D'_M)$. By definition of process refinement there exists a model $M \in \mathbf{Mod}(D)$ such that $F'_M \not\subseteq F_M$ or $D'_M \not\subseteq D_M$. As $F'_M = \mathcal{A}(M)^{*\checkmark} \times \mathcal{P}(\mathcal{A}(M)^{\checkmark})$, see the proof of Lemma 2, we know that $F'_M \subseteq F_M$ holds. Therefore, we know that $D'_M \not\subseteq D_M$. As $D_M = \emptyset$, there exists a trace $t \in D_M$ not ending with $\checkmark$, as the healthiness condition $D3$ of the failures/divergences model asserts that for any trace $u' = u \frown \langle\checkmark\rangle \in D_M$ also $u \in D_M$. Applying healthiness condition $D1$ we obtain $t \frown t' \in D_M$ for all $t' \in \mathcal{A}(M)^{*\checkmark}$. Hence, d'_m is not divergence free, as D'_M violates **C.2** – contradiction to (D,P) divergence free.

3.4 Refinement with Change of Signature

Until now we have analyzed properties of specifications based on a notion of refinement over the same signature. Often, in a refinement step, it is the case that the signature changes. In this section we sketch a first idea of how a theory of refinement with change of signature might look like. In a CSP-CASL institution as discussed in [15], the *semantics* of refinement under change of signature is merely a consequence of our above Definition 1. The aim of this section, however, is to provide a *proof rule* for such a setting. For simplicity, we consider embeddings only and restrict ourselves to the traces model $\mathcal{T}$.

A subsorted CASL signature $\Sigma = (S, TF, PF, P, \leq)$ consists 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 [17] for details.

Definition 4. *We say that a signature $\Sigma = (S, TF, PF, P, \leq)$ is* ***embedded into*** *a signature $\Sigma' = (S', TF', PF', P', \leq')$ if $S \subseteq S'$, $TF \subseteq TF'$, $PF \subseteq PF'$, $P \subseteq P'$, and additionally the following conditions regarding subsorting hold:*

preservation and reflection $\leq = \leq' \cap (S \times S)$.
weak non-extension *For all sorts $s_1, s_2 \in S$ and $u' \in S'$:*
if $s_1 \neq s_2$ and $s_1,\ s_2 \leq' u'$ then there exists $t \in S$ with $s_1,\ s_2 \leq t$ and $t \leq' u$.
sort condition $S' \subseteq \{s' \in S' \mid \exists s \in S : s' \leq' s\}$.

We write $\sigma : \Sigma \rightarrow \Sigma'$ for the induced map from Σ to Σ', where $\sigma(s) = s, \sigma(f) = f, \sigma(p) = p$ for all sort symbols $s \in S$, function symbols $f \in TF \cup PF$ and predicate symbols $p \in P$.

The conditions 'preservation and reflection' and 'weak non-extension' are inherited from the CSP-CASL design, see [19]. The 'sort condition' ensures that reducts are defined, see Lemma 3. In a development process, these conditions allow one to refine the type system by the introduction of new subsorts. Operation symbols and predicate symbols can be added without restriction.

Let $\Sigma = (S, TF, PF, P, \leq)$ be embedded into $\Sigma' = (S', TF', PF', P', \leq')$, then every Σ'-model M' defines a the *reduct* Σ-model $M' \mid_\sigma$, such that $(M' \mid_\sigma)_s = M'(\sigma(s)) = M'_s$, $(M' \mid_\sigma)_f = M'(\sigma(s)) = M'_f$, and $(M' \mid_\sigma)_p = M'(\sigma(s)) = M'_p$ for all sort symbols $s \in S$, function symbols $f \in TF \cup PF$ and predicate symbols $p \in P$. On the alphabet level, the map σ induces an injective map $\sigma^{\mathcal{A}}_{M'} : \mathcal{A}(M' \mid_\sigma) \rightarrow \mathcal{A}(M')$, where $\sigma^{\mathcal{A}}_{M'}([(s,x)]_{\sim_{M'\mid_\sigma}} = [(\sigma^S(s), x)]_{\sim_{M'}}$ – see [19] for the definition of $\sim$. In the following, we will make use of the partial inverse $\hat{\sigma}^{\mathcal{A}}_{M'} : \mathcal{A}(M') \rightarrow ?\ \mathcal{A}(M' \mid_\sigma)$ of this map. In [13] it is shown that, given an injective map, the canonical extension of its partial inverse to trace sets preserves the healthiness conditions in the traces model $\mathcal{T}$. Applying this result to our setting, we obtain:

$$T' \in \mathcal{T}(\mathcal{A}(M')) \Rightarrow \hat{\sigma}^{\mathcal{A}}_{M'}(T') \in \mathcal{T}(\mathcal{A}(M' \mid_\sigma)).$$

This allows us to lift the maps $\hat{\sigma}^{\mathcal{A}}_{M'}$ to the domain of the traces model and we can define:

Definition 5 (Refinement with change of signature). *Let Σ and Σ' be signatures such that Σ is embedded into Σ'. Let $\sigma : \Sigma \rightarrow \Sigma'$ be the induced signature morphism from Σ to Σ'. Let $(d_M)_{M \in I}$ and $(d'_{M'})_{M' \in I'}$ be a families of process denotations over Σ and Σ', respectively.*

$$d_M \rightsquigarrow^{\sigma}_{\mathcal{T}} d'_{M'} \Leftrightarrow I'|_\sigma \subseteq I \wedge \forall M' \in I' : d_{M'|_\sigma} \sqsubseteq_{\mathcal{T}} \hat{\sigma}^{\mathcal{A}}_{M'}(d'_{M'}).$$

Here, $I'|_\sigma = \{M'|_\sigma \mid M' \in I'\}$, and $\sqsubseteq_{\mathcal{T}}$ denotes CSP traces refinement. Figure 3 summarize the overall idea of refinement with change of signature. Thanks to the 'sort condition' we have:

Lemma 3. *In the above setting, the elements $\hat{\sigma}^{\mathcal{A}}_{M'}(d'_{M'})$ are defined over the model $\mathcal{T}$.*

Let $Sp = (D, P)$ and $Sp' = (D', P)$ be two specifications where the signature of D is embedded into the signature of D' and the process parts are syntactically identical. We say that there is a *data refinement with hiding* from Sp to Sp' , in signs $Sp \overset{\texttt{data}}{\rightsquigarrow_\sigma} Sp'$, if $\mathbf{Mod}(D')|_\sigma \subseteq \mathbf{Mod}(D)$.

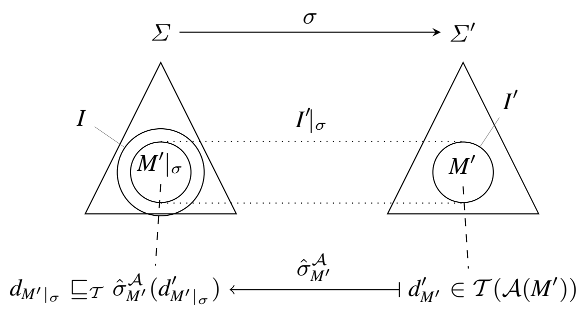
Σ
σ
Σ'
I
$I'|_\sigma$
I'
$M'|_\sigma$
M'
$\hat{\sigma}^{\mathcal{A}}_{M'}$
$d_{M'|_\sigma} \sqsubseteq_{\mathcal{T}} \hat{\sigma}^{\mathcal{A}}_{M'}(d'_{M'|_\sigma})$
$d'_{M'} \in \mathcal{T}(\mathcal{A}(M'))$

Fig. 2. Refinement with change of signature

Theorem 6. *Let $Sp = (D, P)$ and $Sp' = (D', P')$ be* CSP-CASL *specifications, such that the signature of D is embedded into the signature of D'. Then $(D, P) \overset{data}{\rightsquigarrow}_\sigma (D', P)$ and $(D', P) \overset{proc}{\rightsquigarrow}_{\mathcal{T}} (D', P')$ imply $(D, P) \rightsquigarrow^{\sigma}_{\mathcal{T}} (D', P')$.*

Proof. (Sketch) Similar to [14], we prove a *reduct property* by structural induction, namely $traces([\![P]\!]_{M'_{|\sigma}}[a_1/x_1, ..., a_k/x_k]) = \hat{\sigma}^{\mathcal{A}}_{M'}([\![P]\!]_{M'}[\sigma^{\mathcal{A}}_{M'}(a_1)/x_1, ..., \sigma^{\mathcal{A}}_{M'}(a_k)/x_k]))$. Here, the $x_i : s_i$ are free variables in P, and the a_i ranges over $[s_i]_{\sim_{M'|_\sigma}}$, i.e. the set of values in the alphabet generated by the sort symbol s_i. We then prove that this reduct property also applies to least fixed points (in the case of cpo semantics) and unique fixed points (in the case of cms semantics). Let (D, P) have denotations $(d_M)_{M \in \mathbf{Mod}(D)}$, (D', P) have denotations $(d'_{M'})_{M' \in \mathbf{Mod}(D')}$, and (D', P') have denotations $(d''_{M'})_{M \in \mathbf{Mod}(D')}$. The data refinement immediately gives the required model class inclusion. The process refinement yields $d'_{M'|_\sigma} \sqsubseteq_{\mathcal{T}} d''_{M'}$. As $\hat{\sigma}^{\mathcal{A}}_{M'}$ is monotonic w.r.t. traces refinement, and using the reduct property, we obtain $d_{M'|_\sigma} = \hat{\sigma}^{\mathcal{A}}_{M'}(d'_{M'|_\sigma}) \sqsubseteq_{\mathcal{T}} \hat{\sigma}^{\mathcal{A}}_{M'}(d''_{M'})$.

4 Electronic payment system: EP2

In this section we apply the theoretical results presented so far in an industrial setting. The EP2 system is an electronic payment system and it stands for 'EFT/POS 2000', short for 'Electronic Fund Transfer/Point Of Service 2000', is a joint project established by a number of (mainly Swiss) financial institutes and companies in order to define EFT/POS infrastructure for credit, debit, and electronic purse terminals in Switzerland (`www.eftpos2000.ch`). The system consists of seven autonomous entities: CardHolder, Point of Service, Attendant, POS Management, Acquirer, Service Center and Card. These components are centered around an EP2 *Terminal*. These entities communicate with the Terminal and, to a certain extent, with one another via XML-messages in a fixed format. These messages contain information about authorisation, financial transactions, as well as initialisation and status data. The state of each component heavily depends on the content of the exchanged data. Each component is a reactive system defined by a number of use cases. Thus, there are both reactive parts and data parts which need to be modeled, and these parts are heavily intertwined. The EP2 specification consists of 12 documents, each of which describe the different components or some aspect common to the components. The way that the specifications are written is typical of a number of similar industrial application. The specification consists of a mixture of plain English and semi-formal notation. The top level EP2 documents provide an overview of the data involved, while the presentation of further details for a specific type is delayed to separate low-level documents. CSP-CASL is able to match such a document structure by a library of specifications, where the informal design steps of the EP2 specification are mirrored in terms of a formal refinement relation defined in the previous sections. A first modeling approach of the different levels of EP2 in CSP-CASL has been described in [6].

In this section we consider two levels of the EP2 specification, namely: the *architectural level* (ARCH) and the *abstract component level* (ACL). We choose a dialogue between the *Terminal* and the *Acquirer*. In this dialogue, the Terminal and the Acquirer are supposed to exchange initialization information. For presentation purposes, we study here only a nucleus of the full dialogue, which, however, exhibits all technicalities present in the full version.

4.1 Formal refinement in EP2

Our notion of CSP-CASL refinement mimics the informal refinement step present in the EP2 documents: There, the first system design sets up the interface between the components (architectural level), then these components are developed further (abstract component level). Here, we demonstrate in terms of a simple example how we can capture such an informal development in a formal way.

We first specify the data involved using CASL only. The data specification of the architectural level (D_ARCH_GETINIT) consists only of one set of data:

spec D_ARCH_GETINIT =
  sort D_SI_Init
end

In the EP2 system, these values are communicated over channels; data of sort *D_SI_Init* is interchanged on a channel *C_SI_Init* linking the Terminal and the Acquirer. On the architectural level, both these processes just ‘run’, i.e., they are always prepared to communicate an event from *D_SI_Init* or to terminate. We formalize this in CSP-CASL:

ccspec ARCH_INIT =
data D_ARCH_GETINIT
channel $C\_SI\_Init : D\_SI\_Init$
process
  let $Acquirer = EP2Run \quad Terminal = EP2Run$
  in $Terminal\,|[\,C\_SI\_Init\,]|\,Acquirer$
end

Here, $EP2Run = (C\_SI\_Init\ ?\ x : D\_SI\_Init \to EP2Run) \,\Box\, SKIP$. On the abstract component level (D_ACL_GETINIT), data is refined by introducing a type system on messages. In CASL, this is realised by introducing subsorts of *D_SI_Init*. For our nucleus, we restrict ourselves to four subsorts, the original dialogue involves about twelve of them.

spec D_ACL_GETINIT =
  sorts $SesStart, SesEnd, DataRequest, DataResponse < D\_SI\_Init$
  ops $r : DataRequest;\ e : SesEnd$

```
    axioms ∀x : DataRequest; y : SesEnd.¬(x = y)
           ∀x : DataRequest; y : SesStart.¬(x = y)
           ∀x : DataResponse; y : SesEnd.¬(x = y)
           ∀x : DataResponse; y : SesStart.¬(x = y)
end
```

In the above specification, the axioms prevent confusion between several sorts. Using this data, we can specify the ACL level of the Acquirer-Terminal dialogue in CSP-CASL. In the process part the terminal (*TerInit*) initiates the dialogue by sending a message of type *SesStart*; on the other side the Acquirer (*AcqInit*) receives this message. The process *AcqConf* takes the internal decision either to end the dialogue by sending the message *e* of type *SesEnd* or to send another type of message. The Terminal (*TerConf*), waits for a message from the Acquirer, and depending on the type of this message, the Terminal engages in a data exchange. The system as a whole consists of the parallel composition of Terminal and Acquirer.

```
ccspec ACL_INIT =
data D_ACL_GETINIT
channels C_ACL_Init : D_SI_Init
process
    let AcqInit =C_ACL_Init ? session : SesStart → AcqConf
        AcqConf = C_ACL_Init ! e → Skip
                  ⊓ C_ACL_Init ! r → C_ACL_Init ? resp : DataResponse
                      → AcqConf
        TerInit =C_ACL_Init ! session : SesStart → TerConf
        TerConf = C_ACL_Init ? confMess →
                  (if (confMess : DataRequest)
                   then C_ACL_Init ! resp : DataResponse → TerConf
                  else if (confMess : SesEnd) then Skip else Stop)
    in TerInit |[C_ACL_Init]| AcqInit
end
```

Theorem 7. ARCH_INIT $\rightsquigarrow^{\sigma}_{\mathcal{T}}$ ACL_INIT

Proof. Using tool support, we establish this refinement by introducing two intermediate specifications RUN_ARCH and ACL_SEQ:

```
ccspec RUN_ARCH =
 data D_ARCH_GETINIT
 channel C_SI_Init : D_SI_Init
 process EP2Run
end
```

```
ccspec ACL_SEQ =
 data D_ACL_GETINIT
 channels C_ACL_Init : D_SI_Init
 process
     let SeqStart =C_ACL_Init ! session : SesStart → SeqConf
         SeqConf = C_ACL_Init ! e → Skip
                   ⊓ C_ACL_Init ! r
                     → C_ACL_Init ! resp : DataResponse → SeqConf
     in SeqStart
end
```

With CSP-CASL-Prover we proved: ARCH_INIT $=_{\mathcal{T}}$ RUN_ARCH. Now we want to prove that RUN_ARCH $\rightsquigarrow^{\sigma}_{\mathcal{T}}$ ACL_SEQ. To this end, we apply Theorem 6: Using *HETS* [13], we automatically prove the data refinement D_ARCH_GETINIT $\overset{\mathtt{data}}{\rightsquigarrow}_{\sigma}$ D_ACL_GETINIT.

Now, we formed the specification $(\text{D\_ACL\_GETINIT}, P_{\text{ACL\_SEQ}})$, where $P_{\text{ACL\_SEQ}}$ denotes the process part of ACL_SEQ. Next we show in CSP-CASL-Prover that, over the traces model $\mathcal{T}$, this specification refines to ACL_SEQ:

```
theorem Arch_ACL_refinement : "EP2Run <=T SeqStart "
apply(unfold EP2Run_def SeqStart_def)
apply (rule cspT_fp_induct_right[of _  _"Seq_to_Run"])
apply simp_all
apply (induct_tac procName)
...
apply (simp add: cspT_semantics)
apply rule
apply (simp add: in_traces)
apply (auto simp add: D_SI_Init_def SesStart_def  SesEnd_def
                      DataRequest_def DataResponse_def)
...
done
```

Fig. 3. Snippet of proof script for RUN_ARCH $\rightsquigarrow^{\sigma}_{\mathcal{T}}$ ACL_SEQ..

Figure 3 shows a snippet of the proof script for $(\text{D\_ACL\_GETINIT}, P_{\text{ACL\_SEQ}}) \overset{\mathtt{proc}}{\rightsquigarrow}_{\mathcal{T}}$ ACL_SEQ. We first unfold the definitions of `EP2Run` and `SeqStart`. Next, we apply (metric) fixed point induction on the rhs and make a case distinction over the process names, here encoded as `induct_tac procName`. After rewriting and decomposing both of the processes we compute the trace semantics and check that there is indeed an inclusion of traces.

[18] proves ACL_INIT $=_{\mathcal{F}}$ SEQ_INIT. As stable failure equivalence implies trace equivalence, we obtain ACL_INIT $=_{\mathcal{T}}$ SEQ_INIT. Figure 4 summarizes this proof structure.

4.2 Deadlock Analysis of EP2

As ACL_INIT involves parallel composition, it is possible for this system to deadlock. Furthermore, the process *TerConf* includes the CSP process *STOP* within one branch of its conditional. Should this branch of *TerConf* be reached, the whole system will be in deadlock.

The dialogue between the Terminal and the Acquirer for the exchange of initialization messages have been proven to be deadlock free in [18]. Specifically, it has

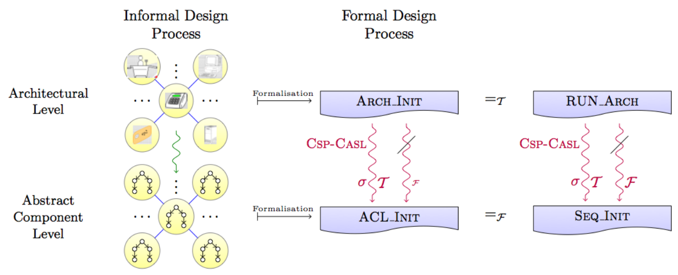

Fig. 4. Refinement in EP2

been proven that the following refinement holds: ACL_SEQ $\overset{\text{proc}}{\rightsquigarrow}_{\mathcal{F}}$ ACL_INIT, where ACL_SEQ is a sequential system. Sequential system are regarded to be deadlock-free.

With our proof method from Section 3.2, we can strengthen this result by actually proving that ACL_SEQ is deadlock free. To this end, we proved with CSP-CASL-Prover that DF $\overset{\text{proc}}{\rightsquigarrow}_{\mathcal{F}}$ ACL_SEQ where DF is the least refined deadlock-free process described in Section 3.2.

4.3 Analysis of Divergence Freeness of EP2

As described in Section 3.3 divergence freeness is best analysed in the model $\mathcal{N}$. The model $\mathcal{N}$ has not yet been implemented in CSP-CASL-Prover. However, using basic step and distributivity laws we have manually shown that ACL_INIT $=_{\mathcal{N}}$ ACL_SEQ and DIVF $\rightsquigarrow_{\mathcal{N}}$ ACL_SEQ, i.e., the EP2 dialogue considered here is divergence free.

5 Conclusions and Future Work

In this paper we have studied various property preserving refinement notions for CSP-CASL. We established proof methods based on decomposition theorems which enable us to reason about refinement using interactive theorem proving. We reduce the analysis of deadlock and divergence freeness to refinement statements.

We showed that our theoretical results apply to a "real world" system. We proved in a systematic way using CSP-CASL-Prover and *HETS* a refinement step from the architectural specification of EP2 to a more detailed one. We proved that at this level of abstraction EP2 is free of deadlock and free of divergence.

Future work will include the extension of our theory on refinement with change of signature to arbitrary signature morphisms as well as the exploration of more "sophisticated" refinement notions for CSP-CASL. In [2], Bidoit et al., present a refinement notion based on observational interpretation of CASL specifications. Following this work we intend to develop observational refinement for CSP-CASL. In the context of EP2 such refinement would be required in order to capture the relations between the more detailed levels.

Acknowledgements The authors would like to thank Liam P. O'Reilly for good support with CSP-CASL-Prover and Erwin R. Catesbeiana (Jr) for numerous, but often divergent discussions on the topic of refinement.

References

1. D.-A. Atiya, S. King, and J. Woodcock. Simpler reasoning about system properties: a proof-by-refinement technique. *Electronic Notes Theoretical Computer Science*, 137(2), 2005.
2. M. Bidoit, D. Sannella, and A. Tarlecki. Observational interpretation of CASL specifications. *Mathematical Structures in Computer Science*, 18(2), 2008.
3. B. Buth, M. Kouvaras, and H. Shi. Deadlock analysis for a fault-tolerant system. In *AMAST'97*, LNCS 1349. Springer, 1997.
4. H. Ehrig and H.-J. Kreowski. Refinement and implementation. In E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors, *Algebraic Foundations of Systems Specification*. Springer, 1999.
5. A. Gimblett. Tool support for CSP-CASL. MPhil Thesis, Swansea University, 2008.
6. A. Gimblett, M. Roggenbach, and H. Schlingloff. Towards a formal specification of an electronic payment systems in CSP-CASL. In *WADT'04*, LNCS 3423. Springer, 2005.
7. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
8. Y. Isobe and M. Roggenbach. Webpage on CSP-Prover. `http://staff.aist.go.jp/y-isobe/CSP-Prover/CSP-Prover.html`.
9. Y. Isobe and M. Roggenbach. A generic theorem prover of CSP refinement. In *TACAS 2005*, LNCS 3440. Springer, 2005.
10. Y. Isobe, M. Roggenbach, and S. Gruner. Extending CSP-Prover by deadlock-analysis: Towards the verification of systolic arrays. In *FOSE 2005*, Japanese Lecture Notes Series 31. Kindai-kagaku-sha, 2005.
11. C. Lüth, M. Roggenbach, and L. Schröder. CCC —the CASL Consistency Checker. In *WADT 2004*, LNCS 3423. Springer, 2005.
12. K. Lüttich and T. Mossakowski. Reasoning support for CASL with automated theorem proving systems. In *WADT 2006*, LNCS 4409, 2007.
13. T. Mossakowski, C. Maeder, and K. Lüttich. The Heterogeneous Tool Set, HETS. In *TACAS 2007*, LNCS 4424. Springer, 2007.
14. T. Mossakowski and M. Roggenbach. Structured CSP – A Process Algebra as an Institution. In *WADT 2006*, LNCS 4409, 2007.
15. T. Mossakowski and M. Roggenbach. An institution for processes and data. In *WADT 2008 – Preliminary Proceedings*, TR-08-15. Universita Di Pisa, 2008.
16. T. Mossakowski, D. Sannella, and A. Tarlecki. A simple refinement language for CASL. In *WADT 2004*, LNCS 3423. Springer, 2004.
17. P. D. Mosses, editor. CASL *Reference Manual*. LNCS 2960. Springer, 2004.
18. L. O'Reilly, Y. Isobe, and M. Roggenbach. CSP-CASL-Prover – a generic tool for process and data refinement. *Electronic Notes in Theoretical Computer Science*, to appear.
19. M. Roggenbach. CSP-CASL – A new integration of process algebra and algebraic specification. *Theoretical Computer Science*, 354, 2006.
20. A. Roscoe. *The theory and practice of concurrency*. Prentice Hall, 1998.
21. H. Shi, J. Peleska, and M. Kouvaras. Combining methods for the analysis of a fault-tolerant system. In *AMAST'98*. Springer, 1999.
22. N. Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4), 1971.