Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 250 (2009) 69–84
www.elsevier.com/locate/entcs

CSP-CASL-Prover: A Generic Tool for Process and Data Refinement

Liam O'Reilly[1] Markus Roggenbach[1]

Swansea University, United Kingdom

Yoshinao Isobe[1,2]

AIST, Tsukuba, Japan

Abstract

The specification language CSP-CASL allows one to model processes as well as data of distributed systems within one framework. In our paper, we describe how a combination of the existing tools HETS and CSP-Prover can solve the challenges that CSP-CASL raises on integrated theorem proving for processes and data. For building this new tool, the automated generation of theorems and their proofs in Isabelle/HOL plays a fundamental role. A case study of industrial strength demonstrates that our approach scales up to complex problems.

Keywords: Process Algebra, Algebraic Specification, Theorem Proving, Functional Programming.

1 Introduction

Distributed computer applications like flight booking systems, web services, and electronic payment systems such as the EP2 standard [2], require parallel processing of data. Consequently, these systems have concurrent aspects (e.g. deadlock-freedom) as well as data aspects (e.g. functional correctness). Often, these aspects depend on each other.

In [22], we present the language CSP-CASL, which is tailored to the specification of distributed systems. CSP-CASL integrates the process algebra CSP [7,23] with the algebraic specification language CASL [15]. Its novel aspects include the combination of denotational semantics in the process part and, in particular, loose semantics for

[1] This cooperation was supported by the EPSRC Project EP/D037212/1.
[2] This work was supported by KAKENHI 20500023.

doi:10.1016/j.entcs.2009.08.018

the data types covering both concepts of partiality and sub-sorting. In [5] we apply Csp-Casl to the EP2 standard and demonstrate that Csp-Casl can deal with problems of industrial strength.

Here, we develop theorem proving support for Csp-Casl and show that our approach scales up to practically relevant systems such as the EP2 standard. Csp-Casl comes with a simple, but powerful notion of refinement. Csp-Casl refinement can be decomposed into first a refinement step on data only and then a refinement step on processes. Data refinement is well understood in the Casl context and has good tool support already. Thus, we focus here on process refinement. The basic idea is to re-use existing tools for the languages Casl and Csp, namely for Casl the tool Hets [13] and for Csp the tool Csp-Prover [8,9,10,11], both of which are based on the theorem prover Isabelle/HOL [19]. This re-use is possible thanks to the definition of the Csp-Casl semantics in a two step approach: First, the data specified in Casl is translated into an alphabet of communications, which, in the second step, is used within the processes, where the standard Csp semantics are applied.

The main issue in integrating the tools Hets and Csp-Prover into a Csp-Casl-Prover is to implement – in Isabelle/HOL – Csp-Casl's construction of an alphabet of communications out of an algebraic specification of data written in Casl. The correctness of this construction relies on the fact that a certain relation turns out to be an equivalence relation. [22] shows in terms of a manually proven meta theorem that the alphabet construction works out for a large class of Casl data specifications, which is characterised by the static semantics property 'has local top elements'. In Csp-Casl-Prover, we choose to prove the relation to be an equivalence for each Csp-Casl specification individually. This adds an additional layer of trust: complementing the algorithmic check of a static property, we provide a proof in Isabelle/HOL that the construction is valid. The alphabet construction, the formulation of the justification theorems (establishing the equivalence relation), and their proofs can all be automatically generated.

Closely related to Csp-Casl is the specification language μCRL [4]. Here, data types have loose semantics and are specified in equational logic with total functions. The underlying semantics of the process algebraic part is operational. [1] presents a μCRL-Prover based on the interactive theorem prover PVS. The chosen approach is to represent the abstract μCRL data types directly by PVS types, and to give a subset of μCRL processes an operational semantics. Thanks to μCRL's simple approach to data – neither sub-sorting nor partiality are available – there is no need for an alphabet construction – as it is also the case in Csp-Casl in the absence of sub-sorting and partiality. Concerning processes, Csp-Casl-Prover provides semantics to full Csp.

Our paper is organised as follows: Section 2 introduces the Csp-Casl semantics along with a case study from the EP2 system. Section 3 describes the existing tools which we make use of. The overall architecture of Csp-Casl-Prover is presented in Section 4. First we discuss how to build an alphabet to be used as a parameter for the process type of Csp-Prover. Then we consider how integration theorems can

lift proof obligations. Finally, we show how to implement our approach in Haskell. Section 5 concludes our paper with a case study on how to prove deadlock freedom of a dialog within the EP2 system.

This paper supersedes our publication [20]. The full technical details of this work can be found in [21].

2 CSP-CASL

Csp-Casl [22] is a comprehensive language which combines *processes* written in Csp [7,23] 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 included, such as multiple prefix, the various parallel operators, operators for non-deterministic choice, and operators for 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.

Syntactically, a Csp-Casl specification with name N consists of a data part Sp, 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 – see Figure 1 for an instance of this scheme:

$$\textbf{ccspec } N = \textbf{data } Sp \textbf{ channel } Ch \textbf{ process } P \textbf{ end}$$

2.1 EP2 in CSP-CASL

As a running example, we choose a dialog nucleus of the EP2 system [2], see [5] for further details of the modelling approach. The Csp-Casl specification of this dialog can be seen in Figure 1. In this dialog, the credit card terminal and another component, the so-called acquirer, are supposed to exchange initialisation information over the channel `C_SI_Init`. The messages on this channel can be classified into the groups `SessionStart`, `SessionEnd`, `ConfigDataRequest` and `ConfigDataResponse`. In the modelling of the EP2 dialog we ensure that messages of type `ConfigDataRequest` are different from messages of type `SessionEnd`. The system consists of the parallel composition of the terminal and the acquirer. Should one of these two components be in a deadlock, the whole system will be in deadlock.

The original dialog in EP2 has more possibilities for the data exchange. For simplicity, we present here only the above nucleus. However, we successfully applied our approach to the full dialog.

```
ccspec GetInitialisationData =
  data  sorts SessionStart, SessionEnd,
              ConfigDataRequest, ConfigDataResponse < D_SI_Init
        forall x:ConfigDataRequest; y:SessionEnd . not (x=y)
        ops r: ConfigDataRequest; e: SessionEnd
  channel C_SI_Init: D_SI_Init
  process
  let  Ter_Init = C_SI_Init ! sessionStart: SessionStart -> Ter_ConfigManagement
       Ter_ConfigManagement = C_SI_Init ? configMess
         -> IF (configMess: SessionEnd) THEN SKIP ELSE
            (IF (configMess: ConfigDataRequest) THEN
              C_SI_Init ! response: ConfigDataResponse -> Ter_ConfigManagement ELSE STOP)
       Acq_Init = C_SI_Init ? sessionStart: SessionStart -> Acq_ConfigManagement
       Acq_ConfigManagement =
            C_SI_Init ! e -> SKIP
         |~| C_SI_Init ! r -> C_SI-Init ? response: ConfigDataResponse
            -> Acq_ConfigManagement
  in Ter_Init |[ C_SI_Init ]| Acq_Init
```

Fig. 1. Nucleus of an EP2 dialog.

2.2 CSP-CASL semantics

Semantically, a CSP-CASL specification is a family of process denotations for a CSP process, where each model of the data part Sp gives rise to one process denotation. The language CSP-CASL is generic in the choice of a specific CSP semantics.

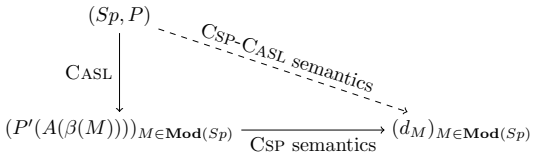

Fig. 2. CSP-CASL semantics.

The semantics of CSP-CASL is defined in a two-step approach [3], see Figure 2. Given a CSP-CASL specification (Sp, P), in the first step we construct for each model M of Sp a CSP process $P'(A(\beta(M)))$. To this end, we define for each model M, which might include partial functions, an equivalent model $\beta(M)$ in which partial functions are totalised. $\beta(M)$ gives rise to an alphabet of communications $A(\beta(M))$. In the second step we point-wise apply a denotational CSP semantics. This translates a process $P'(A(\beta(M)))$ into its denotation d_M in the semantic domain of the chosen CSP model. In the following we sketch the alphabet construction:

A *many-sorted signature* $\Sigma = (S, TF, PF, P)$ consists of a set S of sorts, total functions symbols TF, partial functions symbols PF, and predicate symbols P. Given a many-sorted signature $\Sigma = (S, TF, PF, P)$, a *many-sorted* Σ*-model* M consists of a non-empty carrier set M_s for each sort symbol $s \in S$, a partial function f_M for each function symbol $f \in TF \cup PF$, the function being total for $f \in TF$, and a relation p_M for each predicate symbol $p \in P$. Together with the standard definition of first order logic formulae and their satisfaction, this definition yields the institution $PFOL^{=}$, see [14] for the details.

A *sub-sorted signature* $\Sigma = (S, TF, PF, P, \leq)$ consists of a many-sorted signature (S, TF, PF, P) together with a reflexive and transitive *sub-sort relation* $\leq_S \subseteq S \times S$.

[3] We omit here the syntactic encoding of channels into the data part.

With each sub-sorted signature $\Sigma = (S, TF, PF, P, \leq)$ we associate a many-sorted signature $\hat{\Sigma} = (\hat{S}, \hat{TF}, \hat{PF}, \hat{P})$, which extends the underlying many-sorted signature (S, TF, PF, P) with a total *injection* function symbol $\mathtt{inj}_{\langle s\rangle, s'} \in TF$ for each pair of sorts $s \leq_S s'$, a partial *projection* function symbol $\mathtt{pr}_{\langle s'\rangle, s} \in PF$ for each pair of sorts $s \leq_S s'$, and an unary *membership* predicate symbol $\epsilon^{s}_{s'\langle s'\rangle} \in P$ for each pair of sorts $s \leq_S s'$. *Sub-sorted* Σ*-models* are many-sorted $\hat{\Sigma}$-models satisfying in $PFOL^{=}$ a set of axioms $\hat{J}(\Sigma)$, which prescribe how the injection and projection functions and membership predicates behave [4]. A typical axiom in $\hat{J}(\Sigma)$ is $\mathtt{inj}_{\langle s\rangle, s}(x) \stackrel{e}{=} x$ for $s \in S$. Together with the definition of sub-sorted first order logic formulae and their satisfaction, this definition yields the institution $SubPFOL^{=}$, see [14] for the details.

Given a sub-sorted model M on carrier sets, its strict extension $\beta(M)$ is defined as: $\beta(M)_s = M_s \cup \{\bot\}$ for all $s \in \hat{S}$, where $\bot \notin M_s$ for all $s \in \hat{S}$. We say that a signature $\Sigma = (S, TF, PF, P, \leq)$ has local top elements, if for all $u, u', s \in S$ the following holds: if $u, u' \geq s$ then there exists $t \in S$ with $t \geq u, u'$. Relatively to the extension $\beta(M)$ of a model M for a sub-sorted signature with local top elements, we define an alphabet of communications

$$A(\beta(M)) := (\biguplus_{s \in S} \beta(M)_s)_{/\sim}$$

where $(s, x) \sim (s', x')$ iff either $x = x' = \bot$ and there exists $u \in S$ such that $s \leq u$ and $s' \leq u$; or $x \neq \bot$, $x' \neq \bot$, there exists $u \in S$ such that $s \leq u$ and $s' \leq u$, and for all $u \in S$ with $s \leq u$ and $s' \leq u$ the following holds: $(\mathtt{inj}_{(s,u)})_M(x) = \mathtt{inj}_{(s',u)_M}(x')$ for $s, s' \in S$, $x \in M_s, x' \in M_{s'}$. For signatures with local top elements the relation $\sim$ turns out to be an equivalence relation [22].

2.3 CSP-CASL refinement

Given a denotational Csp model with domain $\mathcal{D}$, the semantic domain of Csp-Casl consists of families of process denotations $d_M \in \mathcal{D}$. Its elements are of the form $(d_M)_{M \in I}$ where I is a class of algebras. As refinement $\rightsquigarrow_{\mathcal{D}}$ we define on these elements

$$(d_M)_{M \in I} \rightsquigarrow_{\mathcal{D}} (d'_{M'})_{M' \in I'} \text{ iff } I' \subseteq I \wedge \forall M' \in I' : d_{M'} \sqsubseteq_{\mathcal{D}} d'_{M'},$$

where $I' \subseteq I$ denotes inclusion of model classes over the same signature, and $\sqsubseteq_{\mathcal{D}}$ is the refinement notion in the chosen Csp model $\mathcal{D}$, e.g., the traces model $\mathcal{T}$, the failures-divergences model $\mathcal{N}$, or the stable-failures model $\mathcal{F}$.

Concerning Csp-Casl refinement, [12] presents the following results:

Theorem 2.1 *For all* (Sp, P), (Sp', P') *holds:*
$(Sp, P) \rightsquigarrow_{\mathcal{D}} (Sp', P') \Longleftrightarrow Sp \rightsquigarrow Sp' \wedge\ (Sp', P) \rightsquigarrow_{\mathcal{D}} (Sp', P')$.

Theorem 2.2 *There exist* (Sp, P), (Sp', P') *such that:*
$(Sp, P) \rightsquigarrow_{\mathcal{D}} (Sp', P') \not\Longrightarrow Sp \rightsquigarrow Sp' \wedge\ (Sp, P) \rightsquigarrow_{\mathcal{D}} (Sp, P')$.

[4] and also define how overloading works.

Here, $Sp \rightsquigarrow Sp'$ denotes CASL refinement in the form of model class inclusion, formally, $\mathbf{Sig}(Sp) = \mathbf{Sig}(Sp')$ and $\mathbf{Mod}(Sp') \subseteq \mathbf{Mod}(Sp)$.

Theorem 2.1 allows the decomposition of a CSP-CASL refinement $(Sp, P) \rightsquigarrow_{\mathcal{D}} (Sp', P')$ into (1) a CASL refinement step from Sp to Sp' and (2) a CSP-CASL refinement step from $(Sp', P) \rightsquigarrow_{\mathcal{D}} (Sp', P')$, where the data part remains constant, namely Sp'. For (1), the tool HETS already offers tool support. Thus, we concentrate here on (2). Concrete: we provide an automatic translation of CSP-CASL refinement over constant data parts into the input language of CSP-Prover.

3 Tools involved

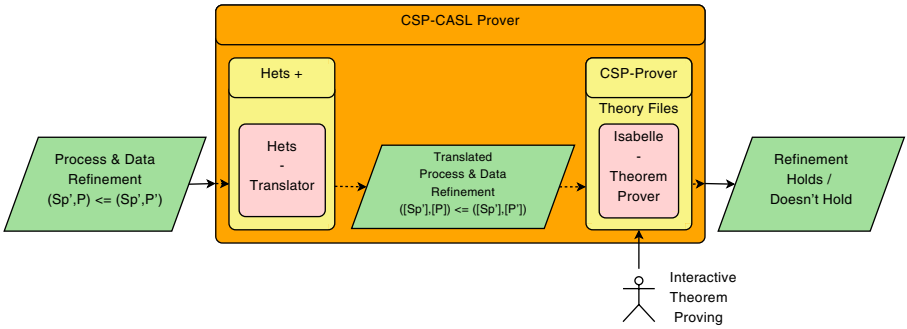

Fig. 3. Diagram of the basic architecture of CSP-CASL-Prover.

Isabelle/HOL [19] is a widely used, interactive theorem prover for Higher Order Logic. Theorems are entered into Isabelle/HOL via *commands*. Isabelle/HOL then displays proof goals to be discharged. To prove a theorem, *proof commands* are issued which transform goals into other goals (or possibly many sub-goals). A goal is discharged if it is transformed into the truth value `True`. A theorem is proven when all of its proof obligations have been discharged. Previously established theorems can be used within further proofs as new proof commands. Proof commands can be combined in various ways to form tactics, which can ease the burden of discharging proof goals. Theory files in Isabelle/HOL consist of scripts of Isabelle commands and proof commands. Commands allow the user to extend the logic, for example, by adding new data structures, types, and function definitions to Isabelle/HOL. This allows the user to accommodate for the particular area of interest.

Hets (the Heterogeneous Tool Set) [13,16] is a parsing, static analysis and proof management tool for various specification languages centred around CASL [15]. One feature of HETS is the ability to translate a specification from one specification language to another specification language, whilst preserving the semantics of the specification. HETS implements various specification languages (seen as logics) and translations (so-called comorphisms) between them. One instance of this mechanism is the translation of CASL specifications into suitable code for use in the theorem prover Isabelle/HOL. CASL *views* trigger HETS to produce proof obligations which

can be discharged by various theorem provers, such as Isabelle/HOL or *SPASS*. HETS is written in the functional programming language Haskell [6]. The HETS code base defines a rich type system which captures mathematical notions such as logics, comorphisms, CASL specifications, theories, etc. HETS makes use of monadic programming in order to simulate states within Haskell.

Csp-Prover [8,9,10,11] is an interactive theorem prover built upon Isabelle/HOL. CSP-Prover is dedicated to refinement proofs over CSP processes, where all CSP operators are supported, including internal and external choice, the various parallel operators, hiding, renaming, as well as recursion. CSP-Prover is generic in the models of CSP that can be used. It can be instantiated with all main CSP models. The traces model $\mathcal{T}$ and the stable-failures model $\mathcal{F}$ are available, while implementations of the stable-revivals model $\mathcal{R}$ and the failure-divergences model $\mathcal{N}$ are underway.

CSP-Prover provides a deep-encoding of CSP within Isabelle/HOL. Consequently, it offers a type `'a proc` (see Section 4.1.2), the type of CSP processes that are built over the alphabet `'a`, where `'a` is an Isabelle/HOL type variable. CSP-Prover comes with a large collection of CSP laws and tactics including CSP *step laws* and *distributivity laws*. One method of proving process equality or refinement without looking into the semantics of the processes is to syntactically rewrite processes using such laws. CSP-Prover's tactics combine such laws to provide powerful proof principles. One typical example is the tactic `cspF_hsf_tac`, which transforms CSP processes to a 'head normal form' over the model $\mathcal{F}$.

4 Implementation of CSP-CASL-Prover

CSP-CASL-Prover uses the existing tools HETS and CSP-Prover. Its architecture is shown in Figure 3. CSP-CASL-Prover takes a CSP-CASL process refinement statement as its input. The CSP-CASL specifications involved are parsed and transformed by CSP-CASL-Prover into a new file suitable for use in CSP-Prover. This file can then be directly used within CSP-Prover to interactively prove if the CSP-CASL process refinement holds. Figure 4 shows the five distinct parts of this file: The first three parts are all automatically generated from the original CSP-CASL specification; the final two parts are dependent on the application. Within the last two parts, CSP-CASL-Prover provides place holder code that the user can fill in and expand.

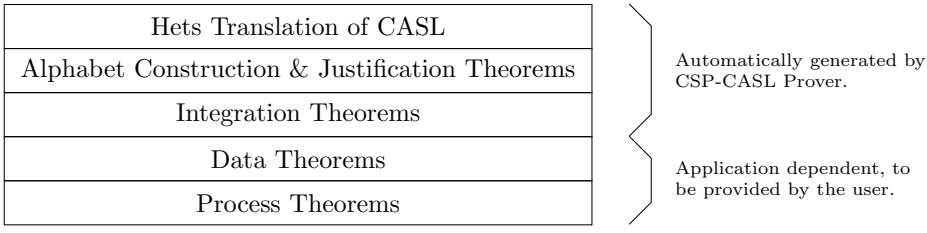

Fig. 4. Structure of a translated CSP-CASL specification using CSP-CASL-Prover.

```
typedecl D_SI_Init
typedecl ConfigDataRequest ...
consts
e :: "SessionEnd"   r :: "ConfigDataRequest"
g__bottom_1 :: "D_SI_Init" ...
g__defined_1 :: "D_SI_Init => bool"
g__defined_2 :: "ConfigDataRequest => bool" ...
g__defined_4 :: "SessionEnd => bool" ...
g__inj_1 :: "ConfigDataRequest => D_SI_Init" ...
g__inj_3 :: "SessionEnd => D_SI_Init" ...
g__proj_1 :: "D_SI_Init => ConfigDataRequest" ...
ga_nonEmpty : "EX x. g__defined_1(x)" ...
ga_notDefBottom : "ALL x. (~ g__defined_1(x)) = (x = g__bottom_1)" ...
Ax1:"ALL x. ALL y. g__defined_2(x) & g__defined_4(y) -->  ~ g__inj_1(x) = g__inj_3(y)"
```

Fig. 5. HETS Encoding for the nucleus of the EP2 specification (Figure 1).

4.1 Alphabet construction

In this section we first discuss a HETS translation from CASL to Isabelle/HOL. Then we describe how to encode the CSP-CASL alphabet construction in Isabelle/HOL.

4.1.1 HETS encoding

We use a semantic preserving encoding provided by HETS to encode the data part of a CSP-CASL specification in Isabelle/HOL[5]. Essentially, HETS produces Isabelle/HOL commands such as `typedecl` (for declaring new non-empty types) and function declarations, followed by axioms which define the properties of such declared types and functions. We discuss here only the signature encoding of $SubPFOL^{=}$ to Isabelle/HOL. CASL sub-sorting and partiality are encoded within Isabelle/HOL by adding undefined elements $\bot_s$ to each sort s, a definedness function D_s for each sort s and injection and projection functions between sorts in the sub-sort relation. The following axioms are then added which control how the bottom elements and definedness functions behave:

(i) $\exists x : s \bullet D_s(x)$ for each $s \in S$,

(ii) $\neg(D_s(x)) \Leftrightarrow (x = \bot_s)$ for each $s \in S$,

(iii) $D_s(f(x_1, \ldots, x_n)) \Leftrightarrow \bigwedge_{i=1\ldots n} D_{s_i}(x_i)$ for each function $f_{\langle s_1,\ldots,s_n\rangle,s} \in TF$,

(iv) $D_s(g(x_1, \ldots, x_n)) \Rightarrow \bigwedge_{i=1\ldots n} D_{s_i}(x_i)$ for each partial function $g_{\langle s_1,\ldots,s_n\rangle,s} \in PF$,

(v) $p(x_1, \ldots, x_n) \Rightarrow \bigwedge_{i=1\ldots n} D_{s_i}(x_i)$ for each predicate symbol $p_{\langle s_1,\ldots,s_n\rangle} \in P$.

Full details of these axioms can be found in [14].

Figure 5 shows part of the encoding[6] that HETS produces for the CSP-CASL specification for the nucleus of the EP2 dialog in Figure 1. According to the sentence translation of [14] the translated axiom from the CSP-CASL specification with the name of `Ax1` has changed slightly from the specification due to the encoding of undefined elements. Now the axiom states that two messages of types `ConfigDataRequest` and `SessionEnd` are never equal if they are both defined.

[5] Currently, the chosen encoding of HETS does not allow for the use of free and generated types, however, this difficulty will be over come in future versions of HETS.

[6] For presentation purposes, we have slightly adapted the naming scheme of HETS.

```
consts
  compare_with_A :: "D_SI_Init => PreAlphabet => bool"
primrec
  compare_with_A_A:  "compare_with_A ax (C_A ay) = (ax = ay)"
  compare_with_A_B:  "compare_with_A ax (C_B by) = (ax = g__inj_1(by))"
...
consts
  eq :: "PreAlphabet => PreAlphabet => bool"
primrec
  eq_A:  "eq(C_A ax) = compare_with_A ax"
  eq_B:  "eq(C_B bx) = compare_with_B bx"
...
```

Fig. 6. Alphabet construction for the nucleus of the EP2 dialog (Figure 1).

4.1.2 Alphabet construction within Isabelle/HOL

Aim of the alphabet construction is to create an alphabet of communications (the new type `Alphabet`) in Isabelle/HOL as set out in Section 2.2. We then use this type within Csp-Prover to form the type `Alphabet proc` of Csp processes over the alphabet of communications.

The alphabet construction of [22] depends on the signature of the data part of a Csp-Casl specification, e.g., on the set of sorts S, see Section 2.2. Hets, however, produces a shallow encoding of Casl only, i.e., there is no type available that captures the set of all sorts of the data part of a Csp-Casl specification. Consequently, it is impossible to give a single alphabet definition within Isabelle/HOL which is generic in the data part of a Csp-Casl specification. Instead, we produce an encoding individually crafted for the data part of any Csp-Casl specification. This crafting follows a systematic approach and is automatically produced by Csp-Casl-Prover in multiple stages: a *construction section* followed by a *justification section*. The *construction section* introduces a new type `PreAlphabet` and defines a relation over this type. The *justification section* is a collection of theorems and proofs which make sure that we are allowed to use the code from the *construction section* in the way we want. The alphabet of communications is then produced using both the type `PreAlphabet` and the relation. In the following we illustrate our construction by an extended example.

The `PreAlphabet` is the disjoint union of all the sorts that Hets produces. The particular code for the creation of the `PreAlphabet` for the nucleus is:

```
datatype  PreAlphabet = C_A D_SI_Init | C_B ConfigDataRequest | C_C ConfigDataResponse
                      | C_D SessionEnd | C_E SessionStart
```

Next a relation called `eq` is defined. This relation takes as parameters two elements of the `PreAlphabet` and checks whether they are equal with respect to the Csp-Casl semantics (this is the relation $\sim$ from Section 2.2).

Figure 6 shows part of the code that is produced for the `eq` relation of the nucleus. Here, auxiliary functions are used to compare each constructor of the datatype `PreAlphabet` with every other constructor. Finally the `eq` relation is defined which makes use of the auxiliary functions. Two elements of the `PreAlphabet` are equal if they are equal in all super-sorts. This is accomplished using the injection functions to test the elements of the `PreAlphabet` at the correct sorts.

The CSP-CASL-semantics requires the relation `eq` to be an equivalence relation. The *justification section* checks that this property holds. The code for checking reflexivity and symmetry is simple. Thus, we focus on the proof of transitivity. The main idea behind this proof is to induct all the variables until only finitely many case distinctions remain. Isabelle/HOL can then automatically solve all of the cases by using some previously proven lemmas which are automatically generated and proven. Figure 7 shows part of the code that is produced to check that the `eq` relation is transitive. We carefully apply induction to the variables `x` and `y` in specific sub-goals by first pulling the sub-goal to the top of the list (using the `prefer` command) and then applying induction to the variable in the first sub-goal. The numbers associated with each `prefer` command are systematically generated by our algorithm.

```
lemma  eq_trans: "[| eq x y ; eq y z |] ==> eq x z"
apply(induct x)
prefer 1 apply(induct y)
prefer 6 apply(induct y)
...
prefer 16 apply(induct y)
prefer 21 apply(induct y)
prefer 1 apply(induct z)
prefer 6 apply(induct z)
...
prefer 116 apply(induct z)
prefer 121 apply(induct z)
apply(auto simp add: g__inj_x_eq_g__inj_y ... g__inj_x_eq_g__inj_y_3)
done
```

Fig. 7. Proof of transitivity of the `eq` relation.

We illustrate this proof idea by a concrete example. Consider the sub-sort structure shown in Figure 8 where the functions shown are the injections functions which HETS provides[7]. After applying all the necessary induction, one of the resulting proof obligations is $x \sim y \wedge y \sim z \Rightarrow x \sim z$, where x, y and z are variables of the types S, T and U, respectively. Expanding the definition of $x \sim z$ yields two new sub-goals: $inj\_S\_U(x) = z$ and $inj\_S\_V(x) = inj\_U\_V(z)$. We focus here on proving $inj\_S\_V(x) = inj\_U\_V(z)$. This equation means that x is equal to z in the sort V. Expanding the definition of $x \sim y$ we obtain the equation $inj\_S\_V(x) = inj\_T\_V(y)$. From $y \sim z$ we obtain $inj\_T\_V(y) = inj\_U\_V(z)$. These two facts together yield $inj\_S\_V(x) = inj\_U\_V(z)$. This proves one part of the goal, the other can be proven in a similar way using the fact that the functions we use are injections (these axioms are provided by HETS). Isabelle/HOL can carry

[7] We use the notation of $\sim$ in place of the Isabelle function `eq`.

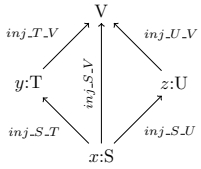

Fig. 8. Example of a possible sub-sort structure with injection functions.

out all these proofs fully automatically, provided the simplifier is enriched with the right injection lemmas, see the last but one line of Figure 7 i.e. `apply(auto simp add: g__inj_x_eq_g__inj_y ... g__inj_x_eq_g__inj_y_3)`. The ideas presented above are automatised in the form of algorithms, see Figure 9 for an algorithm which produces the theorem and proof of transitivity of the `eq` relation.

```
Let n = Number of Sorts in the Specification.
output lemma eq_trans: "[| eq x y; eq y z |] ==> eq x z"
output apply(induct x)
for i = 1 to n   {output prefer (i*n)+1 apply(induct y)}
for i = 1 to n^2 {output prefer (i*n)+1 apply(induct z)}
output apply(auto simp add:‘{all Inject, all Decomp}’)
output done
```

Fig. 9. Algorithm for producing the theorem and proof of transitivity of the `eq` relation.

Now that the justification theorems have been established and proven, we can construct the alphabet of communications in a general way using the following specification independent code (see Section 4.3):

```
instance PreAlphabet::eqv
by intro_classes

defs (overloaded) preAlphabet_sim_def : "x ~ y == eq x y"

instance PreAlphabet::equiv
apply(intro_classes)
apply(unfold preAlphabet_sim_def)
apply(rule eq_refl)
apply(rule eq_trans, auto)
apply(rule eq_symm, simp)
done

types Alphabet = "PreAlphabet quot"
```

4.2 Integration theorems

CSP processes communicate within the alphabet of communications. As the alphabet of communications is a quotient, CSP processes actually communicate equivalence classes. Arguing about the elements of the communications alphabet can therefore be difficult. However, the CSP-CASL-semantics asks only three different questions on the alphabet of communications, see [22]. The most prominent is the test whether two elements of the alphabet of communications are equal or not. This test, for example, is used when two processes synchronise.

In order for the end-user to be able to easily argue on the CSP-CASL process part they need to be able to easily test whether two equivalence classes are equal or not. To facilitate this, CSP-CASL-Prover provides integration theorems which allow tests on the alphabet of communications to be lifted back to tests on the data from the HETS encoding. Figure 10 shows an example of one such integration theorem from the nucleus of the EP2 dialog.

The integration theorem of Figure 10 states that two equivalence classes, which are based on the type "data request" (as they have the form `C_B x`), are equal if and only if their underlying elements of the pre-alphabet are equal in their top most sort

```
lemma int_theorem: "(class(C_B t1) = class(C_B t2)) = (g__inj(t1) = g__inj(t2))"
apply(simp add: quot_equality)
apply(unfold preAlphabet_sim_def)
apply(auto simp add: g__inj_x_eq_g__inj_y ... g__inj_x_eq_g__inj_y_3)
done
```

Fig. 10. Example of an integration theorem and it's proof.

(i.e. `D_SI_Init`). These data theorems and their proofs are automatically generated by algorithms.

Proof practice shows that with these integration theorems available, reasoning about the behavioural aspects of a CSP-CASL specification becomes as easy (or challenging) as reasoning on data and processes separately, where reasoning on processes usually depends on theorems concerning data.

4.3 Dependencies

The following table shows the dependencies of the pre-alphabet construction, justification theorems and the integration theorems on the different elements of the data part of a CSP-CASL specification. $C(D)$ denotes that the construction is dependent on the parameter in the column heading. $T(D)$ denotes that formulation of the theorem statement is dependent on the parameter in the column heading, while $T(I)$ expresses that the theorem statement is independent of the parameter in the column heading. Similarly $P(\_)$ expresses the dependencies of the proofs on the parameter in the column heading.

	Signature	# of Sorts	Sub-sort Structure
Pre-Alphabet Construction	$C(D)$	$C(D)$	$C(D)$
eq_Reflexivity	$T(I)$ / $P(I)$	$T(I)$ / $P(I)$	$T(I)$ / $P(I)$
eq_Symmetry	$T(I)$ / $P(D)$	$T(I)$ / $P(D)$	$T(I)$ / $P(I)$
eq_Transitivity	$T(I)$ / $P(D)$	$T(I)$ / $P(D)$	$T(I)$ / $P(D)$
Integration Theorems	$T(D)$ / $P(D)$	$T(D)$ / $P(D)$	$T(D)$ / $P(D)$

The reflexivity property of the `eq` relation is completely independent of the specification whereas the proof of symmetry relies only on the number of sorts and the proof of transitivity relies on the number of sorts and the sub-sort structure (indirectly). The integration theorems are the most dependent on the specification.

4.4 A prototypical implementation of CSP-CASL-Prover

In this section we discuss our prototypical implementation of CSP-CASL-Prover. [3] provides parser and static analysis support for CSP-CASL within the framework of HETS. Here, CSP-CASL specifications are represented as values of Haskell data types, for instance processes are represented by the data type:

```
data PROCESS
    = Skip Range | Stop Range | GeneralisedParallel PROCESS EVENT_SET PROCESS Range
    ... deriving (Eq, Ord, Show)
```

```
-- Make a PreAlphabet Domain Entry from a list of sorts
mkPreAlphabetDE :: [SORT] -> DomainEntry
mkPreAlphabetDE sorts =
    (Type {typeId = preAlphabetS, typeSort = [isaTerm], typeArgs = []},
     map (\sort -> (mkVName (mkPreAlphabetConstructor sort),
          [Type {typeId = convertSort2String sort, typeSort = [isaTerm], typeArgs = []}])
    ) sorts )
```

Fig. 11. Generation of the type `PreAlphabet`

```
-- Add the symmetry theorem and proof to an Isabelle Theory
addSymmetryTheorem :: [SORT] -> IsaTheory -> IsaTheory
addSymmetryTheorem sorts isaTh =
    let numSorts = length(sorts)
        name = symmetryTheoremS
        x = mkFree "x"
        y = mkFree "y"
        thmConds = [binEq_PreAlphabet x y]
        thmConcl = binEq_PreAlphabet y x
        inductY = concat (map (\i -> [Prefer (i*numSorts+1), Apply (Induct "y")])
                  [0..(numSorts-1)])
        proof' = IsaProof{proof=[Apply (Induct "x")]++inductY++[Apply Auto],end=Done}
    in addThreomWithProof name thmConds thmConcl proof' isaTh
```

Fig. 12. Generation of a justification theorem: proof of symmetry.

The result of the static analysis is a CSP-CASL signature and a list of CSP-CASL sentences. Such CSP-CASL sentences are either CASL formulae or process equations in CSP, see [17,18] for the justification to consider CSP-CASL as a logic. This representation can then be automatically translated into a theory file for Isabelle/HOL using the function:

```
transCCTheory :: (CspCASLSign, [Named CspCASLSentence]) -> Result IsaTheory
```

Here, the type `Result` is a monad provided by the HETS code base for the purpose of collecting together the various results of code analysis. `IsaTheory` is a type representing the abstract syntax of Isabelle/HOL, which HETS then pretty prints.

The translation of the data part is already implemented in HETS: Sub-sorting and partiality are encoded in Isabelle/HOL as required by the CSP-CASL semantics. Thus, our translation is mainly concerned with the encoding of the CSP-CASL semantics and the translation of process equations, for which we present here selected code examples.

The Haskell implementation of the pre-alphabet construction (described in Section 4.1.2) is shown in Figure 11. Here, `DomainEntry` is the type in HETS which represents Isabelle/HOL's `datatype` command. Given a list of sort symbols, each sort symbol gives rise to an alternative constructor, where the symbol's name is used as a part of the respective constructor.

The Haskell code producing a justification theorem and its proof is shown in Figure 12: Given a list of sorts and the current Isabelle/HOL theory we add a new theorem and its proof. Here, we first build the formula to be proven, namely $x \sim y \Longrightarrow y \sim x$. In the code we represent $x \sim y$ as `binEq_PreAlphabet x y`. To this end we produce variables x and y in the abstract Isabelle syntax, form the lhs

```
spec D_ACL_GetInitialisation =
    sorts SessionStart, SessionEnd, ConfigDataRequest, ConfigDataResponse < D_SI_Init
    forall x:ConfigDataRequest; y:SessionEnd . not (x=y)
    ops r: ConfigDataRequest; e: SessionEnd
end
ccspec sequential_system =
  data D_ACL_GetInitialisation
  channels C_SI_Init: D_SI_Init
  process
 let Abstract =
        C_SI_Init ! sessionStart: SessionStart -> Loop
     Loop =   C_SI_Init ! e -> SKIP
           |~| C_SI_Init ! r -> C_SI_Init ! response: ConfigDataResponse -> Loop
  in Abstract
end
```

Fig. 13. Csp-Casl specification of a sequential system.

and rhs of the implication and finally state the implication. Then we build up the proof script which consists of one induction on `x`, which is followed by a sequence of inductions on `y` and rearranging of the proof goals using the proof command `prefer`. Finally, the proof command `auto` is added, and the proof concluded with the proof command `done`.

The instantiation of type classes as well as the generation of the integration theorems uses the same techniques as demonstrated above.

We conclude with an example from the translation of Csp process definitions:

```
transProcess :: PROCESS -> Term
transProcess pr = case pr of
    Skip _ -> cspProver_skipOp
    GeneralisedParallel p es q _ ->
      cspProver_general_parallelOp (transProcess p) (transEventSet es) (transProcess q)...
```

Here, we perform a case distinction on the form of the process and produce the corresponding abstract syntax. `cspProver_skipOp` and `cspProver_general_parallelOp` are values in the abstract syntax of Hets representing the Csp-Prover skip and general parallel operators respectively.

5 Proof of deadlock freedom of EP2

As an application of Csp-Casl-Prover, we prove deadlock freedom in an industrial setting. Our approach is to prove that, in the stable failures model $\mathcal{F}$, the nucleus (see Figure 1 and Section 2.1) is a refinement of the sequential system shown in Figure 13. Here, we have an `Abstract` process that sends a `SessionStart` value and then enters a loop. The `Loop` process either sends a `SessionEnd` message and terminates, or it sends a `ConfigDataRequest` message followed by a `ConfigDataResponse` message and then repeats the loop. `Loop` chooses internally, which of these two branches is taken. As this system has no parallelism it is impossible for it to deadlock. Process refinement within stable failures model preserves deadlock freedom. Hence if we can show that the EP2 nucleus is indeed a refinement of the sequential system, then the EP2 nucleus is guaranteed to be deadlock free.

Figure 14 shows the respective refinement proof in Csp-Casl-Prover (we actu-

```
theorem ep2: "Abs_System =F System"
apply (unfold System_def Abs_System_def)
apply (rule cspF_fp_induct_left[of _ "Abs_System_to_System"])
apply (simp_all)
apply (induct_tac p)
apply (tactic {* cspF_hsf_tac 1 *} | rule cspF_decompo |
       auto simp add: csp_prefix_ss_def image_iff inj_on_def)+
done
```

Fig. 14. Proof of deadlock freedom of the nucleus (see Figure 1).

ally show more, namely that both systems are equivalent). This refinement proof involves recursive process definitions. These are first unfolded, then (metric) fixed point induction is applied. A powerful tactic provided by Csp-Prover finally discharges the proof obligation. This proof also scales up to the full EP2 dialog.

6 Summary and future work

We have shown how to combine the tools Hets and Csp-Prover into a proof tool for Csp-Casl. The main challenges turned out to be the encoding of Csp-Casl's alphabet construction in Isabelle/HOL as well as the automated generation of integration theorems. The alphabet construction turns a many-sorted algebra into a flat set of communications. The integration theorems translate questions on the alphabet of communications back into the language of many-sorted algebra. In both cases, we have devised algorithms and Haskell implementations that – taking a Csp-Casl specification as their input – produce the required types, functions, theorems, and proofs in Isabelle/HOL. A case study on the EP2 system, nearly fully automatically translated, demonstrates that our approach scales up on problems of industrial strength. Future work will include the completion of Csp-Casl-Prover's implementation, analysing further dialogs of EP2 as well as further case studies on distributed computer applications.

Acknowledgement

Thanks to Temesghen Kahsai for his work on decomposition theorems for Csp-Casl refinement and also to Erwin R. Catesbeiana (jr) for his valuable insights into the very nature of electronic payment systems.

References

[1] B. Badban, W. Fokkink, J.F. Groote, J. Pang, and J. van de Pol. Verification of a sliding window protocol in μCRL and PVS. *Formal Aspects of Computing*, 17(3):342–388, 2005.

[2] *eft/pos 2000 Specification, version 1.0.1.* EP2 Consortium, 2002.

[3] Andy Gimblett. Tool support for CSP-CASL, 2008. MPhil Thesis, Swansea University.

[4] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. In *Algebra of Communicating Processes '94*, Workshops in Computing Series, pages 26–62. Springer, 1995.

[5] A. Gimblett, M. Roggenbach, and H. Schlingloff. Towards a formal specification of an electronic payment system in CSP-CASL. In *WADT 2004*, LNCS 3423, pages 61–78. Springer, 2005.

[6] Haskell homepage.
`http://www.haskell.org/`.

[7] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[8] Y. Isobe and M. Roggenbach. A generic theorem prover of CSP refinement. In *TACAS 2005*, LNCS 3440, pages 108–123. Springer, 2005.

[9] Y. Isobe and M. Roggenbach. A complete axiomatic semantics for the CSP stable-failures model. In *CONCUR'06*, LNCS 4137, pages 158–172. Springer, 2006.

[10] Yoshinao Isobe and Markus Roggenbach. CSP-Prover - A proof tool for the verification of scalable concurrent systems. *JSSST (Japan Society for Software Science and Technology) Computer Software*, 25:85–92, 2008.

[11] Yoshinao Isobe and Markus Roggenbach. Proof Principles of CSP – CSP-Prover in Practice. In *LDIC 2007*, pages 425 – 442. Springer, 2008.

[12] Temesghen Kahsai and Markus Roggenbach. Refinement notions for CSP-CASL. In *WADT 2008*, LNCS, to appear. Springer, 2009.

[13] T. Mossakowski, C. Maeder, and K. Lüttich. The Heterogeneous Tool Set, HETS. In *TACAS 2007*, LNCS 4424, pages 519–522. Springer, 2007.

[14] T. Mossakowski. Relating CASL with other specification languages: the institution level. *Theoretical Computer Science*, 286(2):367–475, 2002.

[15] P. Mosses, editor. CASL *Reference Manual*. LNCS 2960 (IFIP Series). Springer, 2004.

[16] T. Mossakowski. Heterogeneous specification and the heterogeneous tool set. Technical report, Universitaet Bremen, 2005. Habilitation thesis.

[17] T. Mossakowski and M. Roggenbach. Structured CSP – A Process Algebra as an Institution. In *WADT 2006*, LNCS 4409, pages 92–110, 2007.

[18] T. Mossakowski and M. Roggenbach. An institution for processes and data. In *WADT 2008 – Preliminary Proceedings*, TR-08-15. Universita Di Pisa, 2008.

[19] T. Nipkow, L.C. Paulon, and M. Wenzel. *Isabelle/HOL*. LNCS 2283. Springer, 2002.

[20] L. O'Reilly, Y. Isobe, and M. Roggenbach. CSP-CASL-Prover – Tool integration and algorithms for automated proof generation. In *CALCO-jnr 2007*. University of Bergen, Febuary 2008.

[21] Liam O'Reilly. Integrating Theorem Proving for Processes and Data, 2008. MPhil Thesis, Swansea University.

[22] M. Roggenbach. CSP-CASL - A new integration of process algebra and algebraic specification. *Theoretical Computer Science*, 354(1):42–71, 2006.

[23] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1998.