CSP-CASL-Prover — Tool integration and algorithms for automated proof generation

Liam O'Reilly[1], Yoshinao Isobe[2], Markus Roggenbach[1*]

[1] Swansea University, United Kingdom
[2] AIST, Tsukuba, Japan

Abstract. The specification language Csp-Casl allows one to model data as well as processes 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.

1 Introduction

Distributed computer applications like flight booking systems, web services, and electronic payment systems such as the EP2 standard [ep202], 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 [Rog06], we present the language Csp-Casl, which is tailored to the specification of distributed systems. Csp-Casl integrates the process algebra Csp [Hoa85,Ros98] with the algebraic specification language Casl [Mos04]. Its novel aspects include the combination of denotational semantics in the process part and, in particular, loose semantics for the data types covering both concepts of partiality and sub-sorting. In [GRS05] 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 [MML07] and for Csp the tool Csp-Prover [IR05,IR06], both of which are based on the theorem prover Isabelle/HOL [NPW02]. 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

* This cooperation was supported by the EPSRC Project EP/D037212/1.

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. Although this has been proven to hold under certain conditions, we chose to prove this fact for each Csp-Casl specification individually. This adds an additional layer of trust. It turns out that the alphabet construction, the formulation of the justification theorems (establishing the equivalence relation), and also the proofs of these theorems can be automatically generated.

Closely related to Csp-Casl is the specification language μCRL [GP95]. 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. [BFG$^+$05] presents - on the fly, as the focus of the paper is on protocol verification – the prototype of a μCRL-Prover based on the interactive theorem prover PVS [ORS92]. The chosen approach is to represent the abstract μCRL data types directly by PVS types, and to give a subset of μCRL processes, namely the linear process equations, an operational semantics in terms of labelled transition systems. 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 provides semantics to full Csp by reusing the implementation of various denotational Csp semantics in Csp-Prover.

Our paper is organized 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. Section 5 discusses in detail how we build a single alphabet which can be used as a parameter for the process type in Csp-Prover. Then we consider how integration theorems can lift proof obligations on the alphabet back onto proof obligations over the data from a Csp-Casl specification. In Section 7 we analyze which parts of our Isabelle code are specification dependent. Section 8 finishes our paper with a case study on how to prove deadlock freedom of a dialog within the EP2 system.

2 CSP-CASL

Csp-Casl [Rog06] is a comprehensive language which combines *processes* written in Csp [Hoa85,Ros98] with the specification of *data types* in Casl [Mos04]. 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, communication over channels. Concerning Casl fea-

tures, the full language is available to specify data types, namely many-sorted first order logic with sort-generation constraints, partiality, and sub-sorting.

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 [ep202], see [GRS05] for further details of the modelling approach. In this dialog, the credit card terminal and another component, the so-called acquirer, are supposed to exchange initialization information over the channel `C_SI_Init`. The messages on this channel can be classified into `SessionStart`, `SessionEnd`, `ConfigDataRequest` and `ConfigDataResponse`. In order to prove that the dialog is deadlock-free, we need to ensure that messages of type `ConfigDataRequest` are different from messages of type `SessionEnd`. The terminal initiates the dialog by sending a message of type `SessionStart`, see the process `Ter_Init`. The acquirer receives this message, see the process `Acq_Init`. In `Acq_ConfigManagement`, the acquirer then takes the internal decision either to end the dialog by sending the message `e` of type `SessionEnd` or to start a data exchange with the terminal. The terminal, on the other side, waits in the process `Ter_ConfigManagement` for a message from the acquirer. Depending on the type of this message, the terminal ends the dialog with `SKIP`, engages in a data exchange, or executes the deadlock process `STOP`. The system consists of the parallel composition of terminal and acquirer. Should one of these two components be in a deadlock, the whole system will be in deadlock.

The original dialog in EP2 has many more possibilities for the data exchange. To this end, it involves 11 sorts, but it exhibits the same structure. For simplicity, we present here only the above nucleus. However, we successfully applied our approach to the full dialog. Our proof on deadlock freedom (see Section 8) scales up from the nucleus to the real version.

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 definition of the language CSP-CASL is generic in the choice of a specific CSP semantics. For example, all denotational CSP models mentioned in [Ros98] are possible parameters.

```
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.

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 totalized. $\beta(M)$ 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 – see [Rog06] for the full details. The purpose of the alphabet construction is to turn a CASL model into an alphabet of communications. CASL models are defined in two steps: First, we define what a model over a many-sorted signature is. Using this concept we then define what a model over a sub-sorted signature is.

A *many-sorted signature* $\Sigma = (S, TF, PF, P)$ consists of

- a set S of sorts,
- two $S^* \times S$-sorted families $TF = (TF_{w,s})_{w \in S^*, s \in S}$ and $PF = (PF_{w,s})_{w \in S^*, s \in S}$ of *total function symbols* and *partial function symbols*, respectively, such that $TF_{w,s} \cap PF_{w,s} = \emptyset$ for each $(w, s) \in S^* \times S$, and
- a family $P = (P_w)_{w \in S^*}$ of *predicate symbols*.

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

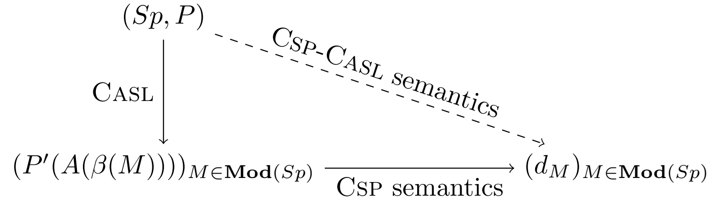

Fig. 2. Csp-Casl semantics.

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 $s \in S$,
- a partial function $(f_{w,s})_M : M_w \to M_s$ for each function symbol $f \in TF_{w,s} \cup PF_{w,s}$, the function being total for $f \in TF_{w,s}$, and
- a relation $(p_w)_M \subseteq M_w$ for each predicate symbol $p \in P_w$.

Together with the standard definition of first order logic formulae and their satisfaction, this definition yields the institution $PFOL^=$, see [Mos02] 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* $\leq_S \subseteq S \times S$. With each sub-sorted signature $\Sigma = (S, TF, PF, P, \leq)$ we associate a many-sorted signature $\hat{\Sigma} = (\hat{S}, \hat{T}F, \hat{P}F, \hat{P})$, which extends the underlying many-sorted signature (S, TF, PF, P) with

- a total *injection* function symbol $\texttt{inj} : s \to s'$ for each pair of sorts $s \leq_S s'$,
- a partial *projection* function symbol $\texttt{pr} : s' \to ?s$ for each pair of sorts $s \leq_S s'$, and
- an unary *membership* predicate symbol $\epsilon^s_{s'} : s'$ 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, projection, and membership behave[4]. A typical axiom in $\hat{J}(\Sigma)$ is $\texttt{inj}_{s,s}(x) \stackrel{e}{=} x$ 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 [Mos02] 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 $s \in \hat{S}$. The strict extension is uniquely determined. 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

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

[4] and also define how overloading works.

where $(s, x) \sim (s', x')$ iff either

- $x = x' = \perp$ and there exists $u \in S$ such that $s \leq u$ and $s' \leq u$,

or

- $x \neq \perp$, $x' \neq \perp$, 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 [Rog06].

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 $\mathcal{D}$. Concerning *data refinement*, we directly obtain the following characterisation:

$$(Sp, P) \rightsquigarrow_{\mathcal{D}} (Sp', P) \quad \text{if} \quad \begin{cases} 1.\ \Sigma(Sp) = \Sigma(Sp'), \\ 2.\ \mathbf{Mod}(Sp') \subseteq \mathbf{Mod}(Sp) \end{cases}$$

The crucial point is that we fix both the signature of the data part and the process P. For *process refinement*, a similar characterisation is obvious:

$$(Sp, P) \rightsquigarrow_{\mathcal{D}} (Sp, Q) \quad \text{if} \quad \begin{cases} \text{for all } M \in \mathbf{Mod}(Sp) : \\ P'(A(\beta(M))) \sqsubseteq_{\mathcal{D}} Q'(A(\beta(M))) \end{cases}$$

CSP-CASL refinement can be decomposed into first a data refinement and then a process refinement[5]:

$$(Sp, P) \rightsquigarrow_{\mathcal{D}} (Sp', P')$$
$$\Longleftrightarrow$$
$$(Sp, P) \overset{\mathtt{data}}{\rightsquigarrow} (Sp', P) \;\wedge\; (Sp', P) \overset{\mathtt{proc}}{\rightsquigarrow}_{\mathcal{D}} (Sp', P')$$

Here, $\overset{\mathtt{data}}{\rightsquigarrow}$ and $\overset{\mathtt{proc}}{\rightsquigarrow}_{\mathcal{D}}$ represent data refinement and process refinement, respectively. Data refinement does not deal with the process part at all. Thus, to prove data refinement we can re-use the existing tool support for CASL. Consequently, we focus in this paper on tool support for process refinement.

[5] Note that the order of the decomposition is essential: $(Sp, P) \rightsquigarrow_{\mathcal{D}} (Sp', P') \not\Rightarrow (Sp, P) \overset{\mathtt{proc}}{\rightsquigarrow}_{\mathcal{D}} (Sp, P') \;\wedge\; (Sp, P') \overset{\mathtt{data}}{\rightsquigarrow} (Sp', P')$.

3 Tools involved

Csp-Casl-Prover makes appropriate re-use of existing technology and tools. In this section we will explain what these tools are and what they do.

3.1 Isabelle/HOL

Isabelle/HOL [NPW02] is a widely used, generic interactive theorem prover for Higher Order Logic. Theorems are entered into Isabelle/HOL via *commands*. Isabelle/HOL then displays proof goals which need to be discharged. For example the command `theorem T1: "a+b = b+a"` creates a theorem with the name `T1` and a goal of `a+b = b+a`.

To prove such 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 consist of scripts of Isabelle commands and proof commands. Such theory files can use theorems, proofs, data structures and functions written in other theory files. This brings in a concept of modularity to Isabelle/HOL.

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.

For example, the command `datatype Num = N nat | I int` adds a new data type with the name `Num`. This creates a new type which is the sum of natural numbers and integers. Here, `N` and `I` are user chosen type constructors while `nat` and `int` are the built in types of natural numbers and integers, respectively. Such a datatype declaration comes with a built in induction tactic within Isabelle/HOL. In our example of `Num`, this induction tactic simplifies to a complete case distinction.

The commands that Isabelle/HOL offers are able to create new data structures, functions, relations, etc. Theorems can then be used to prove properties of data structures, functions and relations, while proof commands can use the definition of such structures, functions and relations. For example, we define a new function `plus:: Num => Num => Num` such that a natural number will be returned only if both arguments are natural numbers, else an integer will be returned. The definition of `plus` is as expected. The following Isabelle/HOL code proves our new function `plus` to be commutative:

```
theorem comm: "plus a b = plus b a"
apply(induct_tac a)
```

The first line sets up the theorem with the name `comm` and states that `plus` is commutative. The second line applies the induction tactic on the variable `a`. This simplifies to a finite case distinction over our sum type, i.e. `a` can have the form

`N nat` or `I int`. Hence, after application of the induction tactic `induct_tac`, the following two sub-goals are shown:

```
goal (theorem (comm), 2 subgoals):
 1. !!nat. plus (N nat) b = plus b (N nat)
 2. !!int. plus (I int) b = plus b (I int)
```

Here `b`, `nat`, and `int` are variables where `nat` and `int` are locally bound in each subgoal (indicated by the `!!` symbol). Further induction on the variable `b` along with simplification proves our theorem.

3.2 HETS

HETS (the Heterogeneous Tool Set) [MML07] is a parsing, static analysis and proof management tool for various specification languages centred around Casl [Mos04].

One of the features of HETS is the ability of translating a specification from one specification language into a specification from another language, while preserving its semantics. An important instance of this is the translation of Casl specifications into suitable code for use in the theorem prover Isabelle/HOL. In our setting we use HETS as an input/output tool, loading specifications written in Casl and encoding them into Isabelle/HOL code. This translation process is non-trivial and Csp-Casl-Prover exploits this functionality heavily.

3.3 CSP-Prover

Csp-Prover [IR05,IR06] is a theorem prover built upon Isabelle/HOL. Csp-Prover is dedicated to refinement proofs over Csp processes. It is generic in the models of Csp that can be used. It can be instantiated with all main Csp models. The trace model $\mathcal{T}$ and the stable-failures model $\mathcal{F}$ are available, while implementations of the stable-revivals model $\mathcal{R}$ and 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` (which is used within Section 5.2), the type of Csp processes that are built over the alphabet `'a`, where `'a` is an Isabelle/HOL type variable.

Csp-Prover supports two proof methods, namely syntactical and semantical proofs. Syntactical proofs transform the syntax of Csp processes into equivalent Csp processes until syntactical identity is reached. Semantical proofs evaluate the denotational semantics of Csp processes and compare the denotations.

Csp-Prover comes with a large collection of Csp laws and tactics. Csp-Prover tactics combine these laws to powerful proof principles. One typical example is the tactic `cspF_hsf_tac`, which transforms Csp processes to a 'head normal form' over the model $\mathcal{F}$.

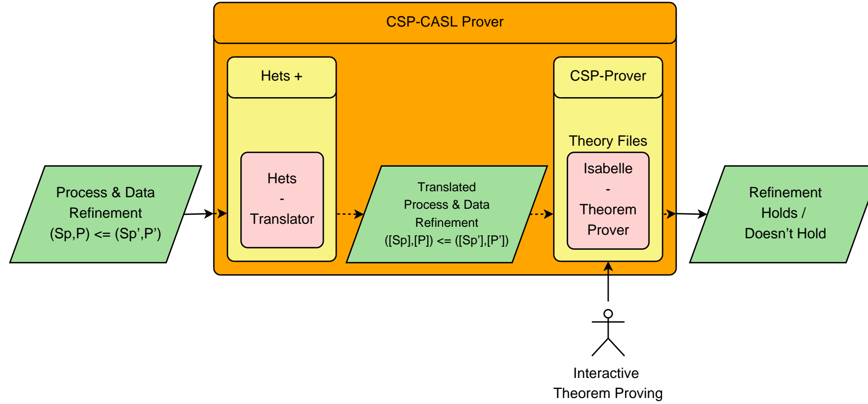

Fig. 3. Diagram of the basic architecture of Csp-Casl-Prover.

4 Basic architecture of CSP-CASL-Prover

Csp-Casl-Prover uses the existing tools HETS and Csp-Prover discussed in Sections 3.2 and 3.3. Its proposed architecture is shown in Figure 3. The overall idea is that 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. For example, dead lock freedom of a system of processes can be proven using such a refinement statement, see Section 8 for details.

Csp-Casl-Prover re-uses the existing functionality of HETS in order to produce part of the file that will be used as input to Csp-Prover. We take the data part of a Csp-Casl specification and translate this into Isabelle/HOL via HETS. This generates (in general) several types in Isabelle/HOL, which need to be transformed into one alphabet to become the parameter of the Csp-Prover process type `'a proc`. This is expressed in Figure 3 by HETS being labelled as "Hets +", which represents the extra encoding that needs to be done. This is discussed in more detail Section 5.2.

The final form of the file which is produced by Csp-Casl-Prover (i.e. HETS and the extra encoding) is labelled as "Translated Processes and Data Refinement" in Figure 3. Figure 4 shows how this file is split up into five distinct parts. The first three parts can all be automatically generated from the original Csp-Casl specification. The final two parts are dependent on the application. Csp-Casl-Prover provides place holder code that the user can fill in and expand.

The first part of the file shown in Figure 4 "Hets Translation of CASL" is the direct encoding of the data part of the Csp-Casl specification which is produced by HETS – see Section 5.1. The second part "Alphabet Construction & Justi-

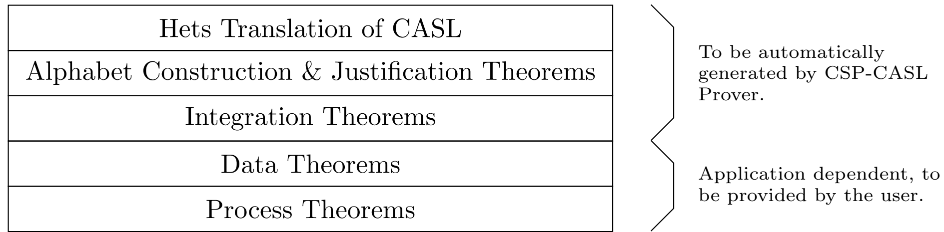

Fig. 4. Structure of a translated Csp-Casl specification using Csp-Casl-Prover.

fication Theorems" provides the Csp-Casl semantics, namely the alphabet of communications, over which Csp processes can be constructed – see Section 5.2. The third part "Integration Theorems" provides the user with a mechanism to lift proof obligations on processes to proof obligations on data in the HETS encoding only – see Section 6. These Integration Theorems are crucial in keeping the final proof of the process refinement small, readable and manageable – see Section 8 for an example. The forth part is where the user shall write auxiliary theorems and proofs which are helpful for the specific refinement to be proven. The final part is where the user shall provide the proof of the refinement between the processes.

5 Alphabet construction

In this section we first discus the encoding that is produced by HETS. Then we describe how to encode the alphabet construction in Isabelle/HOL.

5.1 HETS encoding

We use HETS in order to encode the data part of a Csp-Casl specification[6]. Essentially, HETS produces Isabelle/HOL commands such as `typedecl` and function declarations, followed by axioms which define the properties of such declared types and functions. Casl sub-sorting and partiality are encoded within Isabelle/HOL by adding undefined elements to each sort and by providing injection and projection functions between sorts in the sub-sort relation. Full details of this encoding can be found in Chapter 4 of [Mos02].

Figure 5 shows part of the encoding[7] that HETS produces for the Csp-Casl specification for the nucleus of the EP2 dialog in Figure 1.

HETS produces one `typedecl` for each sort declared in the Csp-Casl specification. A `typedecl` command extends the signature by a new type which is

[6] 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.
[7] For the purposes of a clear presentation in the paper, we have slightly adapted the naming scheme of HETS.

```
typedecl D_SI_Init
typedecl D_SI_Init_ConfigDataRequest ...
consts
e :: "D_SI_Init_SessionEnd"   r :: "D_SI_Init_ConfigDataRequest"
g__bottom_1 :: "D_SI_Init" ...
g__defined_1 :: "D_SI_Init => bool"
g__defined_2 :: "D_SI_Init_ConfigDataRequest => bool"
g__defined_4 :: "D_SI_Init_SessionEnd => bool" ...
g__inj_1 :: "D_SI_Init_ConfigDataRequest => D_SI_Init"
g__inj_3 :: "D_SI_Init_SessionEnd => D_SI_Init" ...
g__proj_1 :: "D_SI_Init => 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).

assumed to be non-empty. After introducing all types for messages, the constants `e` and `r` are declared. Their type is the translated version of the sort from the specification, i.e. `D_SI_Init_SessionEnd` and `D_SI_Init_ConfigDataRequest`, respectively. Then constants representing an undefined element of each sort are declared. The constant `g__bottom_1` represents the undefined element of type `D_SI_Init`. This is where the strict encoding of the models is produced - see Section 2.2. Next, functions are declared to capture definedness, injection and projection functions. This is followed by axioms that control how these function behave, including the axioms of $\hat{J}(\Sigma)$ from Section 2.2 which describe the encoding of the sub-sorted signature into a many-sorted signature. Besides this, the axioms state that there is a single unique undefined element in each sort - see axiom `ga_notDefBottom` – and that each sort has at least one defined element – see axiom `ga_nonEmpty`. Full details of these axioms can be found in [Mos02]. Finally, the original axiom from the Csp-Casl specification has been added with the name of `Ax1`. This axiom changes slightly from the specification because of the encoding of undefined elements. Now the axiom states that two messages of types `D_SI_Init_ConfigDataRequest` and `D_SI_Init_SessionEnd` are never equal if they are both defined.

5.2 Alphabet construction within Isabelle/HOL

The goal 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 with Csp-Prover to form the type `Alphabet proc` of Csp processes over this alphabet of communications.

As HETS produces a shallow encoding of Casl, it is impossible to give a single alphabet definition within Isabelle/HOL. To overcome this obstacle, we

```
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(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).

produce an encoding which is specifically crafted, however in a systematic way for each data part of a Csp-Casl specification. This encoding can automatically be produced by an algorithm that we report on in this paper.

The algorithm generates the code in multiple stages. First the algorithm produces the *construction section* followed by the *justification section*. The *construction section* creates a new type what we call `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. Then the alphabet of communications is produced using both the type `PreAlphabet` and the relation.

The *construction section* consists of a new data-type called the `PreAlphabet` which 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 D_SI_Init_ConfigDataRequest
                      | C_C D_SI_Init_ConfigDataResponse
                      | C_D D_SI_Init_SessionEnd
                      | C_E D_SI_Init_SessionStart
```

Next a relation called `eq` is defined. This relation takes as parameters two elements of the `PreAlphabet` and checks weather 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 data-type `PreAlphabet` with every other constructor. These auxiliary functions are used in order to make use of primitive recursion in Isabelle/HOL. Finally the `eq` relation is defined. Basically, 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. 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[8] . After applying 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 out all these proofs fully automatically, provided the simplifier is en-

[8] We use the notation of $\sim$ inplace of the Isabelle function `eq`.

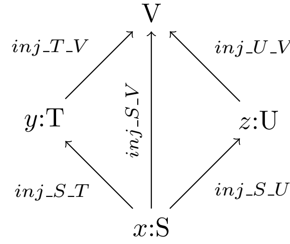

Fig. 8. Example of a possible sub-sort structure with injection functions.

riched with the right injection axioms, see the last but one line `apply(auto simp add: g__inj_x_eq_g__inj_y ... g__inj_x_eq_g__inj_y_3)` of Figure 7.

Figure 9 shows part of the 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.

Finally, after the *justification section*, the alphabet of communications is constructed:

```
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"
```

First we instantiate `PreAlphabet` as the class `eqv` which allows us to define a relation $\sim$. Then we define this relation in terms on the `eq` function. In the next step we instantiate `PreAlphabet` as the class `equiv`, which comes with proof obligations that the $\sim$ relation is indeed an equivalence relation. Finally we create a type synonym called `Alphabet` as the quotient of `PreAlphabet`

6 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, Csp-Casl-semantics asks only three different questions on the alphabet of communications, see [Rog06]. 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. 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.

```
lemma integration_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.

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 (i.e. `D_SI_Init`). Such data theorems and their proofs can be 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.

7 Dependencies

The following table shows the dependencies of the pre-alphabet construction and the integration theorems. $T(D)$ denotes that the theorem is dependent on the parameter in the column heading, while $T(I)$ expresses that the theorem is independent of the parameter in the column heading, and similar $P(_)$ expresses the dependencies of the proofs on the parameter in the column heading.

	Specification	# of Sorts	Sub-sort Structure
Pre-Alphabet Construction	$T(D)$ / $P(D)$	$T(I)$ / $P(D)$	$T(D)$ / $P(D)$
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	$T(D)$ / $P(D)$	$T(I)$ / $P(D)$	$T(D)$ / $P(D)$

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. All these proofs can be automatically generated by algorithms.

8 Proof of deadlock freedom of EP2

```
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. 11. Csp-Casl specification of a sequential system.

As an application of CSP-CASL-Prover we show how to prove deadlock freedom in an industrial setting. Here we prove deadlock freedom of the nucleus as shown in Figure 1. We have also proven deadlock freedom of the full EP2 dialog: the proof script scales up.

Our approach is to prove that, in the stable failures model $\mathcal{F}$, the nucleus is a refinement of the sequential system shown in Figure 11. 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, the EP2 nucleus is guaranteed to be deadlock free.

For our refinement proof we apply the algorithms discussed in this paper on both the EP2 nucleus as well as on the sequential system specification. Adding the integration theorems to Isabelle/HOL's simplifier set then allows us to prove deadlock freedom as shown in Figure 12 (we actually 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 from CSP-Prover finally discharges the proof obligation. The whole proof script involves syntactic proof techniques only.

```
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. 12. Proof of deadlock freedom of the nucleus (see Figure 1).

9 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 managed to come up with an algorithm that – take 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, for the moment carried out manually, demonstrates that our approach scales up on problems of industrial strength.

Future work will include the implementation of the algorithms described as well as further case studies on distributed computer applications. Another direction of work is to consider a semi-deep encoding of Casl. In such a setting the justification theorem which states that the relation `eq` is an equivalence relation becomes specification independent.

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

[BFG+05] 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.

[ep202] *eft/pos 2000 Specification, version 1.0.1.* EP2 Consortium, 2002.

[GP95] 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.

[GRS05] 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.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[IR05] Y. Isobe and M. Roggenbach. A generic theorem prover of CSP refinement. In *TACAS 2005*, LNCS 3440, pages 108–123. Springer, 2005.

[IR06] 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.

[MML07] T. Mossakowski, C. Maeder, and K. Lüttich. The Heterogeneous Tool Set, Hets. In *TACAS 2007*, LNCS 4424, pages 519–522. Springer, 2007.

[Mos02] T. Mossakowski. Relating Casl with other specification languages: the institution level. *Theoretical Computer Science*, 286(2):367–475, 2002.

[Mos04] P. Mosses, editor. Casl *Reference Manual.* LNCS 2960 (IFIP Series). Springer, 2004.

[NPW02] T. Nipkow, L.C. Paulon, and M. Wenzel. *Isabelle/HOL.* LNCS 2283. Springer, 2002.

[ORS92] S. Owre, J.M. Rushby, and N. Shankar. PVS: A prototype verification system. In *CADE-11*, LNAI 607, pages 748–752. Springer, 1992.

[Rog06] M. Roggenbach. CSP-CASL - A new integration of process algebra and algebraic specification. *Theoretical Computer Science*, 354(1):42–71, 2006.

[Ros98] A.W. Roscoe. *The theory and practice of concurrency.* Prentice Hall, 1998.