Electronic Communications of the EASST
Volume 62 (2013)

Specification, Transformation, Navigation
Special Issue dedicated to Bernd Krieg-Brückner
on the Occasion of his 60th Birthday

On the whereabouts of CSP-CASL – A survey

Andy Gimblett, Temesghen Kahsai, Liam O'Reilly, Markus Roggenbach

21 pages

On the whereabouts of CSP-CASL – A survey

Andy Gimblett[1], **Temesghen Kahsai**[2], **Liam O'Reilly**[1], **Markus Roggenbach**[1]

[1] Swansea University, UK [2] The University of Iowa, USA

Abstract: CSP-CASL is but one of the many languages for which Bernd Krieg-Brückner (BKB) had a great deal of influence throughout its development process: from the initial idea of working towards an integration of the process algebra CSP with the algebraic specification language CASL, to the design of the concrete syntax, and also to tool support for CSP-CASL, where the theorem prover Isabelle should provide the common platform. In all this, BKB provided inspiration and guidance, funding, and also a helping hand when needed.

This paper provides a survey on the technology developed so far for CSP-CASL, covering results of a theoretical nature, an industrial case study, theorem proving support as well as a testing approach. In honour of BKB's 60th birthday, this survey documents what has become out of one of BKB's visions.

Keywords: CSP, CASL, CSP-CASL, Modelling, Verification, Theorem Proving, Testing.

1 Introduction

Distributed computer applications like flight booking systems, web services, and electronic payment systems such as the EP2 standard [EP208] involve the parallel processing of data. Consequently, these systems exhibit concurrent aspects (e.g., deadlock-freedom) as well as data aspects (e.g., functional correctness). Often, these aspects depend on each other. The algebraic specification language CASL alone can deal with data aspects, while the process algebra CSP is quite capable of modelling concurrent systems. The mutual dependencies between processes and data, however, require a new type of language: e.g., CSP-CASL [Rog03, Rog06]. The novel aspects of CSP-CASL 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.

CSP-CASL is equipped with a refinement notion based on refinements developed in the context of the single languages CSP and CASL. For system development one is often 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 are then preserved over the design steps.

Based on previous publications, this paper provides a survey on the technology currently available for CSP-CASL. Section 2 presents the language and the basic ideas underlying its semantics, [Rog03, Rog06]. We then review a major industrial case study, namely, how to model the electronic payment system EP2 in CSP-CASL, [GRS05]. Section 4 reports on parsing and statically analysing CSP-CASL, [Gim08]. CSP-CASL refinement [KR09, Kah10] is the topic of Section 5: Besides its obvious use for system development, refinement is the fundamental means to analyse systems within the CSP-CASL context, e.g., in order to check a specification for deadlock-freedom. Section 6 documents the construction of CSP-CASL-Prover [O'R08, OIR09], which offers theorem proving support for checking CSP-CASL refinement. With this technology around, we mechanically verify parts of the EP2 system in Section 7: we prove that the formal specifications representing the different layers of the EP2 system are related via refinement, and demonstrate how to prove deadlock-freedom of typical EP2 dialogues. Modelling and formally analysing a system improves the quality of the design process, however, there still is no direct link between specification and implementation. Sections 8 and 9, finally, present a framework for black-box testing from CSP-CASL specifications, [KRS07, KRS08], and a testing framework for EP2 [Kah10].

1.1 Related approaches

As process algebras tend to lack the ability to handle data, there have been various approaches towards the integration of processes and data: [ABR99] provides a survey on such integrations.

For CASL, various reactive extensions have been defined. They include CCS-CASL [SAA01] and CASL-CHARTS [RR00], which combine CASL with reactive systems of a particular kind, the semantics of which are defined in terms of set theory. CASL extensions such as COCASL [MRS03] (being much simpler than full set theory) and CASL-LTL [RAC00] provide more of what one might call a meta-framework: for example, the process algebra CCS has been formalised in both these languages.

Initial or concrete data types underlie LOTOS [ISO89], E-LOTOS [ISO01], and also CSP_M [Ros98]. This has the shortcoming that data needs to be fixed the moment the modelling is done. A step forward in this respect is the specification language μCRL [GP95]. Here, data types have loose semantics and are specified in equational logic with total functions. CASL, and thus CSP-CASL, offers a more expressive logic for the specification of data, namely many-sorted first order logic with sort-generation constraints, partiality, and sub-sorting. On the process side, μCRL employs a branching time semantics based on bisimulation. CSP-CASL offers the choice between the various trace semantics defined in the context of CSP, i.e., CSP-CASL provides a linear time view on systems.

A different approach towards data is taken in CSP-OZ [Fis97]. Here, the object-oriented state-based formalism Object-Z is combined with the process algebra CSP. Its formal semantics is given in terms of CSP's failures-divergences model. In contrast, CSP-CASL's semantics is generic with respect to the underlying CSP model.

While the above approaches, including CSP-CASL, keep data and processes distinct, Circus [WC01, WC02] provides a deeper integration of data specified in Z with processes written in CSP. While this has advantages for methodology, with deep integration it is difficult to re-use existing tools and established theoretical results from the constituent languages.

Our testing approach is closely related to the work by Cavalcanti and Gaudel for CSP [CG07] and Circus [CG11]. While we study an implementation relation from a specification to the system under test (SUT) in the physical world, Cavalcanti and Gaudel make the assumption that the SUT behaves like some unknown specification and characterize refinement between two mathematical objects using a test suite.

2 The design of CSP-CASL

CSP-CASL [Rog03, Rog06, Gim08] 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 available, such as multiple prefix, the various parallel operators, operators for non-deterministic choice, and communication over channels. Concerning CASL features, the full language is available to specify data types, namely many-sorted first order logic with sort-generation constraints, partiality, and 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 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:

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

See below for concrete instances of this scheme. *Semantically*, such a CSP-CASL specification defines a family of process denotations for a CSP process, where each model of the data part *Sp* gives rise to one process denotation.

The various CSP semantics are built relative to a fixed set of communications, where the semantic clauses involve various test functions over this set. To this end, CSP-CASL's semantical construction provides what we call a *data type of communications*, which, besides an alphabet of communications, defines the following functions:

- test on equality for arbitrary CASL terms
  (can two communications synchronise?),
- test on membership for a CASL term concerning a CASL sort
  (does a communication belong to a certain subset of the alphabet of communications?),
- test whether a binary predicate holds between two CASL terms
  (are the terms in a renaming relation?), and
- satisfaction of a CASL first order formula
  (is the formula of the conditional construct true?).

These test functions, living on the alphabet, can be lifted to formulae in CASL. Figure 1 illustrates this setting. This data type of communications make the language CSP-CASL generic in the choice of a specific CSP semantics. The above listed, seemingly small set of test operations allows for all denotational semantics described in [Ros98], namely trace-semantics, failure-divergence-semantics and stable-failure-semantics.

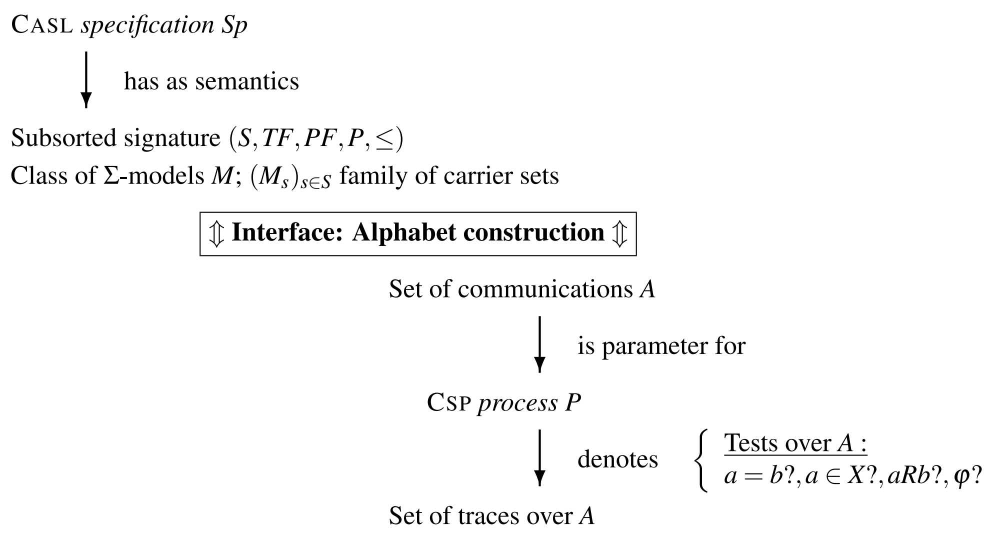
CASL specification Sp
has as semantics
Subsorted signature $(S, TF, PF, P, \leq)$
Class of Σ-models M; $(M_s)_{s \in S}$ family of carrier sets
$\Updownarrow$ Interface: Alphabet construction $\Updownarrow$
Set of communications A
is parameter for
CSP process P
denotes
Tests over A :
$a = b?, a \in X?, aRb?, \varphi?$
Set of traces over A

Figure 1: Translating data into an alphabet of communications.

The data types specified by algebraic specification consist of *many-sorted algebras*. The data type of communications required by the process algebraic semantics is a *one-sorted algebra*. Thus, in order to integrate data into processes, we need to turn a many-sorted algebra into one set of values such that the above described tests are closely connected with the original data type. We illustrate our construction here for the example of many-sorted total algebras:

There are two natural ways to define the alphabet of communications in terms of the carrier sets of a CASL model: union and disjoint union of all carrier sets. To illustrate the effect of both possibilities, consider the following CSP-CASL specification:

$$\begin{array}{lll} \textbf{data} & \textbf{sorts} & S, T \\ & \textbf{ops} & c : S;\ d : T \\ \textbf{process} & \multicolumn{2}{l}{c \to SKIP \ \| \ d \to SKIP} \end{array}$$

Its data part, written in CASL, provides two constants c and d of type S and T, respectively. The process part, written in CSP with CASL terms denoting communications, combines two processes by the synchronous parallel operator, i.e., they have to agree on all actions.

The question is, may c and d synchronise or not? In all the various CSP semantics, c and d synchronise iff they are equal. Now consider two isomorphic CASL models $\mathscr{A}$ and $\mathscr{B}$ of the data part:

$$\mathscr{A}(S) = \{*\},\ \mathscr{A}(T) = \{+\},\ \mathscr{A}(c) = *,\ \mathscr{A}(d) = +$$
$$\mathscr{B}(S) = \mathscr{B}(T) = \{\sharp\},\ \mathscr{B}(c) = \mathscr{B}(d) = \sharp$$

Choosing the union of all carrier sets as alphabet has the effect, that c and d do not synchronise for algebra $\mathscr{A}$ while they synchronise for algebra $\mathscr{B}$. Thus, isomorphic algebras give rise to different behaviour. Therefore, we define the alphabet to be the disjoint union — with the consequence that c and d do not synchronise.

Similar 'experiments' for partiality, subsorting, and the combination of both – see [Rog06] – finally lead to an alphabet construction

$$A(\mathscr{M}) := (\biguplus_{s \in S} \mathscr{M}_s \cup \{\perp\})_{/\sim}.$$

Here $\mathscr{M}$ is a CASL model and S is the set of all sorts declared in the data part. The special element $\perp$ encodes partiality, while $\sim$ is an equivalence relation which – on the alphabet level – deals with subsorting.

3 The first challenge: Modelling EP2

In [GRS05] we describe the formal specification of a banking system using CSP-CASL. The system in question is called EP2, which stands for *EFT/POS 2000*, short for 'Electronic Fund Transfer/Point Of Service 2000'; it is a joint project established by a consortium of (mainly Swiss) financial institutes and companies in order to define EFT/POS infrastructure for credit, debit, and electronic purse terminals in Switzerland[1].

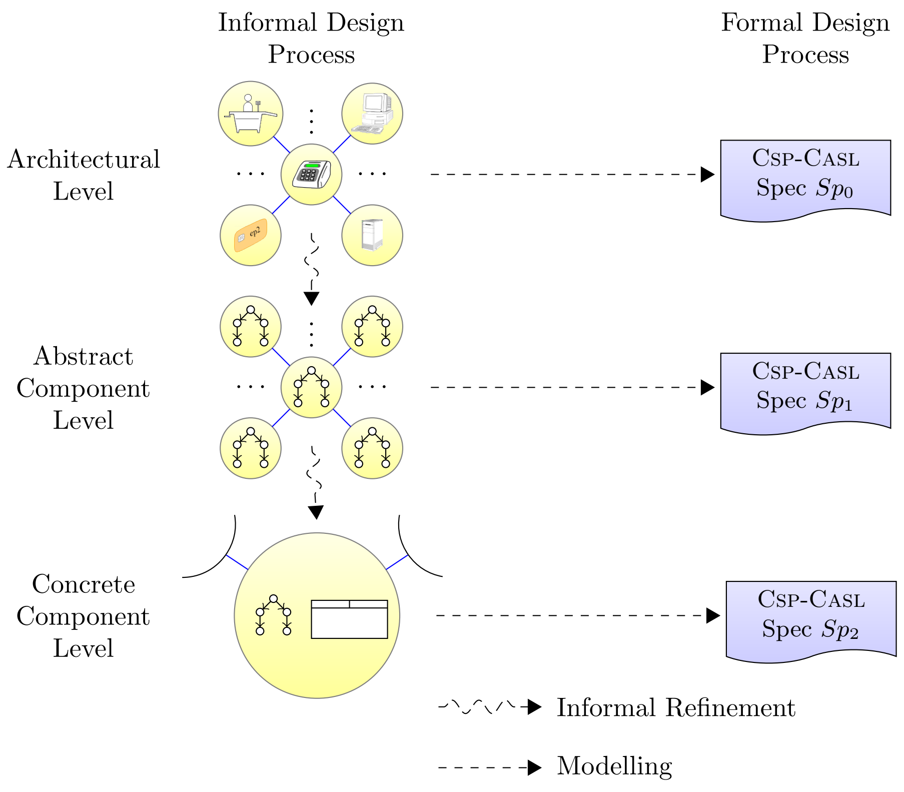

Figure 2: Formalising specifications at various levels of abstraction.

[1] www.eftpos2000.ch

At the architectural level, the system consists of seven autonomous entities centred around the EP2 *Terminal* (see Figure 2, which shows several of these entities), a hardware device concerned with processing card details and authorising financial transactions. The other entities are the *Cardholder* (i.e., customer), *Point of Service/POS* (i.e., cash register), *Attendant*, *POS Management System/PMS*, *Acquirer*, *Service Center*, and *Card*. These entities communicate with the Terminal, and to a certain extent one another, mostly via XML messages over TCP/IP; the messages contain information about authorisation, financial transactions, and initialisation and status data. Each component is thus a reactive system, and there are both reactive parts and data parts which need to be modelled, with these parts heavily intertwined.

The EP2 specification, a typical informal specification as used in industry, consists of twelve documents. These documents are written using a number of different specification notations: plain English; UML-like graphics (use cases, activity diagrams, message sequence charts, class models, etc.); pictures; tables; lists; file descriptions; encoding rules. In these regards it is entirely typical of a modern industrial strength system specification.

With such a document structure, the information required to understand a particular aspect of the system is contained in several different documents, each of which has its own things to say about the situation in question. For example, the `SI-Init` interface between Terminal and Acquirer is documented in the Terminal Specification, the Acquirer Specification, the Interface Specification, and the Data Dictionary. In general, as we found, this results in a specification which is difficult to understand, and which easily leads to inconsistencies and ambiguities.

As such, this is exactly the kind of system we would like to specify formally, with all the associated benefits of tool support that this brings. When [GRS05] was written, such tool support for CSP-CASL was unavailable; nonetheless, we were able to specify in CSP-CASL a number of aspects of the system at varying levels of abstraction, as indicated in Figure 2, and corresponding to different views used in the informal specification documents. In particular, we wrote an architectural specification providing the system overview, and abstract component specifications of several elements including the `SI-Init` interface between Terminal and Acquirer. In doing so, we identified a number of ambiguities and even contradictions in the EP2 specification. For example, the order of communications over the `SI-Init` interface was specified in two contradictory ways in two different documents. Thus, the very act of writing specifications in a formal language required a level of rigour at which such problems were forced into the light.

With our specifications we were then able to perform (relatively simple) proofs on paper — for example, we proved deadlock-freedom of part of our `SI-Init` specification. It quickly became clear, however, that in order to deal with the full complexity of a specification of this sort, tool support would be absolutely essential.

4 From blackboard language to tool support

For quite a while, CSP-CASL existed only as a ‘blackboard language’: while the subject of several papers, its syntax and semantics remained malleable, allowing experimentation and discussion. In order to achieve its full promise, however, it was clear that tool support would be required. The first step towards such tool support was to fix CSP-CASL’s syntax and static semantics, and to implement them within an appropriate tool. This work is described fully in [Gim08].

Element	**Machine readable**	**Pretty printed**	**Precedence**
Named process	$P\,\mathtt{(}t_1\mathtt{,}\ \mathtt{...}\ \mathtt{,}t_n\mathtt{)}$	$P(t_1,\ldots,t_n)$	0
Skip	`SKIP`	SKIP	0
Stop	`STOP`	STOP	0
Div	`DIV`	DIV	0
Run	$\mathtt{RUN\,(}es\mathtt{)}$	$\mathit{RUN}(es)$	0
Chaos	$\mathtt{CHAOS\,(}es\mathtt{)}$	$\mathit{CHAOS}(es)$	0
Term prefix	$t\mathtt{->}p$	$t \rightarrow p$	1
Internal prefix choice	$\mathtt{\vert\tilde{}\vert}\,x\mathtt{::}s\mathtt{->}p$	$\sqcap x :: s \rightarrow p$	1
External prefix choice	$\mathtt{[\,]}\,x\mathtt{::}s\mathtt{->}p$	$\Box x :: s \rightarrow p$	1
Channel send	$c\,\mathtt{!}\,t\mathtt{->}p$	$c!t \rightarrow p$	1
Channel non-deterministic send	$c\,\mathtt{!}\,x\mathtt{::}s\mathtt{->}p$	$c!x :: s \rightarrow p$	1
Channel receive	$c\,\mathtt{?}\,x\mathtt{::}s\mathtt{->}p$	$c?x :: s \rightarrow p$	1
Sequential composition	$p\,\mathtt{;}\,q$	$p\,;q$	2
Internal choice	$p\,\mathtt{\vert\tilde{}\vert}\,q$	$p \sqcap q$	3
External choice	$p\,\mathtt{[\,]}\,q$	$p \Box q$	3
Interleaving	$p\,\mathtt{\vert\vert\vert}\,q$	$p \mid\mid\mid q$	4
Synchronous	$p\,\mathtt{\vert\vert}\,q$	$p \mid\mid q$	4
Generalised parallel	$p\,\mathtt{[\vert}\,es\,\mathtt{\vert]}\,q$	$p[\![es]\!]q$	4
Alphabetised parallel	$p\,\mathtt{[}\,es_1\,\mathtt{\vert\vert}\,es_2\,\mathtt{]}\,q$	$p[\![es_1 \mid\mid es_2]\!]q$	4
Hiding	$p\backslash es$	$p \setminus es$	5
Renaming	$p\,\mathtt{[[}R\mathtt{]]}$	$p[\![R]\!]$	5
Conditional	$\mathtt{if}\ \varphi\ \mathtt{then}\ p\ \mathtt{else}\ q$	$\mathrm{if}\,\varphi\,\mathrm{then}\,p\,\mathrm{else}\,q$	6

Figure 3: Summary of CSP-CASL process terms.

CSP-CASL is presented as a CASL extension language; its syntax and static semantics extend, and refer to, those of CASL; as such, they are formalised in a style matching that of CASL found in [Mos04]. The abstract and concrete syntaxes are written in the normal BNF-like style (see Figure 3 for an informal overview of the concrete syntax of process terms), whereas the static semantics is in the style of *natural semantics* [Kah87]. As a representative example, here is the static semantics rule for a conditional process term:

$$\frac{\begin{array}{cc} & \Sigma,\Lambda,G,L \vdash \mathtt{P}_1 \rhd p,\alpha_p \\ (\Sigma, G\cup L) \vdash \mathtt{F} \rhd \varphi & \Sigma,\Lambda,G,L \vdash \mathtt{P}_2 \rhd q,\alpha_q \end{array}}{\Sigma,\Lambda,G,L \vdash \mathtt{condition\text{-}proc}\ \mathtt{F}\ \mathtt{P}_1\ \mathtt{P}_2 \rhd \mathrm{if}\,\varphi\,\mathrm{then}\,p\,\mathrm{else}\,q,\alpha_p\cup\alpha_q\cup \mathit{sorts}(\varphi)}$$

The rule shows how to devolve the static check on an abstract syntax element for a conditional process term ($\mathtt{condition\text{-}proc\ F\ P_1\ P_2}$), into checks on the condition (a CASL formula `F`) and processes ($\mathtt{P}_1$ and $\mathtt{P}_2$) involved. It operates within some context (a CASL signature Σ; a so-called CSP-CASL signature Λ, containing process and channel declarations; and global and local variable bindings G and L), elements of which then provide context for the devolved rules. Note that the `FORMULA` rule for analysing the text F is found in the CASL static semantics. If the devolved checks succeed, they yield semantic objects representing the formula (φ), process terms (p, q) and the process terms' constituent alphabets (α_p, α_q), which are combined into the semantic objects yielded as the overall product of the rule: an object representing the entire conditional term, and an object representing its constituent alphabet, itself the subject of later checks, at the process equation level.

The syntax and static checks are implemented within the framework of the heterogeneous toolset, HETS [MML07]. CASL sits at the centre of a family of related specification languages – its extensions and sublanguages – and the language HETCASL [Mos05] then provides a framework for *heterogeneous specification* over this family[2]. That is, it enables specification using a mix of languages, rather than requiring all aspects of a system to be specified using the same formalism. HETS then provides tool support for heterogeneous specification in this setting. It consists of a general framework for working with specifications across multiple logics, with facilities for parsing, static analysis, error reporting, translation between logics, and integration with proof tools; logics for the various CASL-family languages (including CSP-CASL) are implemented within this framework.

5 Refinement and property verification in CSP-CASL

Our notions of refinement for CSP-CASL [Rog06, KR09, Kah10] are based on refinements developed in the context of the single languages CSP and CASL. For a denotational CSP model with domain $\mathscr{D}$, the semantic domain of CSP-CASL consists of families of process denotations $d_M \in \mathscr{D}$ over some index set I,

$$(d_M)_{M\in I}$$

where I is a class of CASL models over the same signature. 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 means less non-deterministic choice.

Definition 1 (Model class inclusion) Let $(d_M)_{M\in I}$ and $(d'_{M'})_{M'\in I'}$ be families of process denotations.

$$(d_M)_{M\in I} \rightsquigarrow_{\mathscr{D}} (d'_{M'})_{M'\in I'} \Leftrightarrow I' \subseteq I \wedge \forall M' \in I' : d_{M'} \sqsubseteq_{\mathscr{D}} d'_{M'}.$$

Here, $I' \subseteq I$ denotes inclusion of model classes over the same signature, and $\sqsubseteq_{\mathscr{D}}$ is the refinement notion in the chosen CSP model $\mathscr{D}$. In the traces model $\mathscr{T}$ we have for instance $P \sqsubseteq_{\mathscr{T}} P' :\Leftrightarrow traces(P') \subseteq traces(P)$, where $traces(P)$ and $traces(P')$ are prefixed closed sets of traces. Given CSP-CASL specifications $Sp = (D,P)$ and $Sp' = (D',P')$, by abuse of notation we also write

$$(D,P) \rightsquigarrow_{\mathscr{D}} (D',P')$$

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 characterise situations, where one specification part remains constant. In

[2] In fact, HETCASL is generic in its language family, and could be used as a heterogeneous specification framework in an entirely different setting.

a *data refinement*, only the data part changes:

$$(D,P) \overset{\texttt{data}}{\rightsquigarrow} (D',P') \quad \text{if} \quad \Sigma(D) = \Sigma(D') \text{ and } \mathbf{Mod}(D') \subseteq \mathbf{Mod}(D)$$

In a *process refinement*, the data part is constant:

$$(D,P) \overset{\texttt{proc}}{\rightsquigarrow}_{\mathscr{D}} (D,P') \quad \text{if} \quad \text{for all } M \in \mathbf{Mod}(D) : \ \mathbf{den}([\![P]\!]_M) \sqsubseteq_{\mathscr{D}} \mathbf{den}([\![P']\!]_M)$$

Here, $\mathbf{den}([\![P]\!]_M)$ is the denotation of the process P over the model M in the chosen CSP denotational model. Clearly, both these refinements are special forms of CSP-CASL refinement in general, for which the following holds:

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 that consists of the process part P of Sp and the data part D' of Sp'. For these three specifications holds:*

$$(D,P) \rightsquigarrow_{\mathscr{D}} (D',P')$$
$$\Longleftrightarrow$$
$$(D,P) \overset{\texttt{data}}{\rightsquigarrow} (D',P) \ \textit{and} \ (D',P) \overset{\texttt{proc}}{\rightsquigarrow}_{\mathscr{D}} (D',P')$$

This result forms the basis for the CSP-CASL tool support, see Section 6. In order to prove that a CSP-CASL refinement $(D,P) \rightsquigarrow_{\mathscr{D}} (D',P')$ holds, first one uses proof support for CASL alone, see [MML07], in order to establish $\mathbf{Mod}(D') \subseteq \mathbf{Mod}(D)$. Independent of this, one has then to check the process refinement $\mathbf{den}([\![P]\!]_M) \sqsubseteq_{\mathscr{D}} \mathbf{den}([\![P']\!]_M)$ for all $M \in \mathbf{Mod}(D')$ – see the next section for a discussion on tool support.

For system development one is often interested in liberal notions of refinements, which allow for substantial changes in the design. For system verification, however, it is important that refinement steps preserve properties. This allows the verification of properties on abstract specifications which in general are less complex than the more concrete ones. Here, we show how to perform deadlock-freedom analysis using refinement.

In the CSP context, the stable failures model $\mathscr{F}$ is best suited for deadlock analysis. The model $\mathscr{F}$ 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. Let A be the alphabet. The process *STOP*, which represents deadlock in the CSP context, has

$$traces(STOP) = \{\langle\rangle\}, \qquad failures(STOP) = \{(\langle\rangle, X) \mid X \subseteq A^{\checkmark}\}$$

as its denotation in $\mathscr{F}$, 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[3].

A CSP-CASL specification is deadlock-free, if it enforces all its possible implementations to be deadlock-free. On the semantical level, we capture this idea as follows:

[3] $\checkmark \notin A$ is a special event signalling successful termination, $A^{\checkmark} = A \cup \{\checkmark\}$.

Definition 2 Let $(d_M)_{M \in I}$ be a family of process denotations over the stable failures model, i.e., $d_M = (T_M, F_M) \in \mathscr{F}(A(M))$ for all $M \in I$, where $A(M)$ is the alphabet induced by the model M, see Section 2.

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

Deadlock can be analysed through refinement checking: The most abstract deadlock-free CSP-CASL specification over a CASL signature Σ with a set of sort symbols $S = \{s_1, \dots, s_n\}$ is defined as:

ccspec DF$_S$ =
  data ...declaration of Σ ...
  process $DF = (\bigsqcap_{s:S} x : s \to DF) \sqcap Skip$
end

We observe that DF$_S$ is deadlock-free. This result on DF$_S$ extends to a complete proof method for CSP-CASL deadlock analysis:

Theorem 2 *A* CSP-CASL *specification* (D,P) *is deadlock-free iff* DF$_S \rightsquigarrow_{\mathscr{F}} (D,P)$. *Here S is the set of sorts of the signature of D.*

Similar ideas allow to employ the notion of refinement to check for divergence-freedom, see [KR09].

While in this section we have illustrated refinement notions for CSP-CASL over the same signature, [Kah10] defines refinement notions, proof support and property analysis (deadlock and livelock), which allow for arbitrary change of signature.

6 CSP-CASL-Prover

CSP-CASL-Prover [O'R08, OIR09] exploits the fact that CSP-CASL refinement can be decomposed into first a refinement step on data only and then a refinement step on processes, see the previous section. 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, IR07, IR08], both of which are based on the theorem prover Isabelle/HOL [NPW02]. 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. In CSP-CASL-Prover, we choose to prove the relation $\sim$ – see Section 2 – to be an equivalence relation 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.

The architecture of CSP-CASL-Prover is shown in Figure 4. CSP-CASL-Prover takes a CSP-CASL process refinement statement as its input. The CSP-CASL specifications involved are

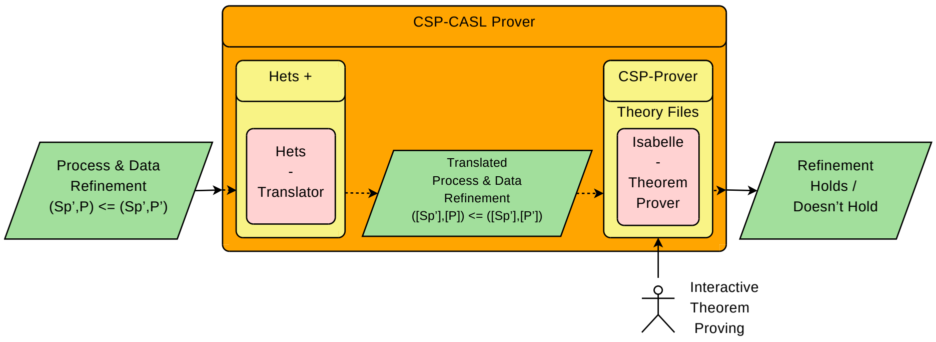
CSP-CASL Prover
Hets +
Hets - Translator
CSP-Prover
Theory Files
Isabelle - Theorem Prover
Process & Data Refinement (Sp',P) <= (Sp',P')
Translated Process & Data Refinement ([Sp'],[P]) <= ([Sp'],[P'])
Refinement Holds / Doesn't Hold
Interactive Theorem Proving

Figure 4: Diagram of the basic architecture of CSP-CASL-Prover.

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.

The alphabet construction discussed in Section 2 depends on the signature of the data part of a CSP-CASL specification, e.g., on the set of sorts S, see Section 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, CSP-CASL-Prover produces an encoding individually crafted for the data part of any CSP-CASL specification.

7 Property verification of EP2

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 initialisation information. For presentation purposes, present specifications at high levels of abstraction only and we study a nucleus of the full dialogue only. This setting, however, already exhibits the full interplay of processes and data. Furthermore, it is a strength of CSP-CASL to be able to capture these abstraction levels at all.

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 how we can capture such an informal development in a formal way, see Figure 5.

We first specify the data involved using CASL only. The data specification of the architectural level (D_ARCH_GETINIT) requires only that there is a set of values available:

spec D_ARCH_GETINIT = **sort** *D_SI_Init* **end**

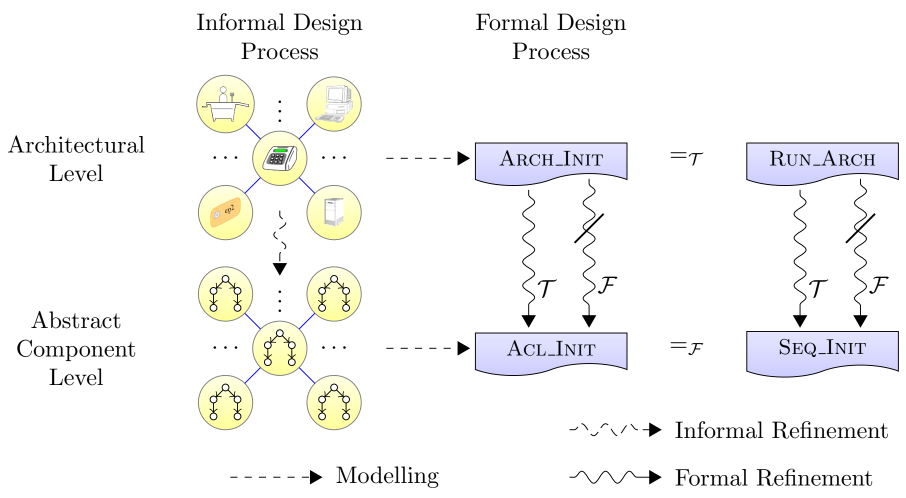

Figure 5: Refinement in EP2.

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 formalise this in CSP-CASL:

ccspec ARCH_INIT =
data D_ARCH_GETINIT
channel *C_SI_Init* : *D_SI_Init*
process
  let *Acquirer* = *EP2Run* *Terminal* = *EP2Run*
  in *Terminal* $|[$ *C_SI_Init* $]|$ *Acquirer*
end

where $EP2Run = (C\_SI\_Init\ ?\ x : D\_SI\_Init \rightarrow EP2Run) \, \Box \, SKIP$. 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 $\forall x : DataRequest;\ y : SesEnd. \neg(x = y)$; $\forall x : DataRequest;\ y : SesStart. \neg(x = y)$;
    $\forall x : DataResponse;\ y : SesEnd. \neg(x = y)$; $\forall x : DataResponse;\ y : SesStart. \neg(x = y)$
end

In the above specification, the axioms prevent confusion between the different sorts. Using this data, we can specify the EP2 system 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. In *AcqConf*, the acquirer 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 3 ARCH_INIT $\rightsquigarrow^{\sigma}_{\mathscr{T}}$ ACL_INIT

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

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

```
ccspec SEQ_INIT =
 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 $=_T$ RUN_ARCH. Now we want to prove that RUN_ARCH $\rightsquigarrow^{\sigma}_{\mathscr{T}}$ SEQ_INIT. Using HETS we showed:
D_ARCH_GETINIT $\overset{\mathtt{data}}{\rightsquigarrow}_{\sigma}$ D_ACL_GETINIT. Now, we formed the specification (D_ACL_GETINIT, $P_{\text{SEQ\_INIT}}$) and showed in CSP-CASL-Prover that, over the traces model $\mathscr{T}$, this specification refines to SEQ_INIT. Here, $P_{\text{SEQ\_INIT}}$ denotes the process part of SEQ_INIT. [OIR09] proves ACL_INIT $=_{\mathscr{F}}$ SEQ_INIT. As stable failure equivalence implies trace equivalence, we obtain ACL_INIT $=_{\mathscr{T}}$ SEQ_INIT. Figure 5 summarises this proof structure. □

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 initialisation messages have been proven to be deadlock-free in [OIR09]. Specifically, it has been proven that the following refinement holds: SEQ_INIT $\overset{\text{proc}}{\rightsquigarrow}_{\mathscr{F}}$ ACL_INIT, where SEQ_INIT is a sequential system. Sequential systems are regarded to be deadlock-free. With our proof method from Section 5, we can strengthen this result by actually proving that SEQ_INIT is deadlock-free. To this end, we proved with CSP-CASL-Prover that $DF_S \overset{\text{proc}}{\rightsquigarrow}_{\mathscr{F}}$ SEQ_INIT, where DF_S is the least refined deadlock-free specification. Details of the various proof can be found in [Kah10].

8 Specification based testing

The fundamental question when dealing with testing based on formal specifications is the following: which are the intentions or properties formulated in the specification text? In the end, each test case reflects some intentions described in the specification.

In order to study the above question for CSP-CASL, in [KRS07] we undertake a specification exercise in modelling a one-bit calculator. It has two input buttons and can compute the addition function only. Whenever one of the buttons is pressed on the calculator, the integrated control circuit displays the corresponding digit in the display. After pressing a second button, the corresponding addition result is displayed and the calculator returns to its initial state.

In a first high-level specification we might want to abstract from the control flow and just specify the interfaces of the system:

ccspec BCALC0 =
  data sort *Number*
    ops $0, 1 : Number$;
      $\_\_ + \_\_ : Number \times Number \to ? Number$
  channel *Button, Display* : *Number*
  process $P_0 = (?x : Button \to P_0) \sqcap (!y : Display \to P_0)$
end

Relatively to BCALC0, the process

$$T_0 = Button!0 \to Button!0 \to STOP$$

encodes *'left open'* behaviour: The specification says that the process P_0 internally decides if it will engage in an event of type *Button* or in an event of type *Display*. Thus, if the environment offers to press the zero button twice, a legitimate implementation of BCALC0 can in one experiment refuse to engage in this behaviour (by deciding internally that initially it engages in an event of type *Display*), while in a second run it engages in it. A more refined version requires that the first displayed digit is echoing the input, and the second displays the result of the computation:

ccspec BCALC1 =
  data sort *Number*
    ops $0, 1 : Number$;
      $\_\_ + \_\_ : Number \times Number \to ? Number$
  channel *Button, Display* : *Number*
  process $P_1 = ?x : Button \to Display!x \to ?y : Button \to Display!(x + y) \to P_1$
end

Relatively to BCALC1, now the process T_0 encodes *'forbidden'* behaviour as it does not alternate between pressing buttons and display of digits. The process

$$T_1 = Button!0 \rightarrow Display!0 \rightarrow Button!1 \rightarrow Display!1 \rightarrow STOP$$

however, again represents *'left open'* behaviour: We haven't yet specified the arithmetic properties of addition, i.e., it is undecided yet if $0+1=1$ or $0+1 \neq 1$. This underspecification can be resolved by adding suitable axioms:

ccspec BCALC2 =
  data sort $Number$
    ops $0, 1 : Number;$
      $\_\_ + \_\_ : Number \times Number \rightarrow ?\, Number$
    axioms $0+0=0;\ 0+1=1;\ 1+0=1$
  channel $Button, Display : Number$
  process $P_2 = ?x : Button \rightarrow Display!x \rightarrow ?y : Button \rightarrow Display!(x+y) \rightarrow P_2$
end

Relatively to BCALC2, T_1 represents *'required'* behaviour.

We encode the intention of a test case with respect to a specification by a colouring scheme. Intuitively, green test cases reflect *required* behaviour of the specification. Red test cases reflect *forbidden* behaviour of the specification. A test is coloured yellow if it depends on an *open design decision*, i.e., if the specification does neither require nor disallow the respective behaviour. Internal non-determinism and underspecification are the sources of yellow tests.

Formally, a test case is just a CSP-CASL process in the same signature as the specification, which may additionally use first-order variables ranging over communications. The *colour* of a test case T with respect to a CSP-CASL specification (Sp, P) is a value $c \in \{red, yellow, green\}$. [KRS07] provides, besides a mathematical definition of test colouring, also a syntactic characterisation of test colouring in terms of refinement. This enables the use of CSP-CASL-Prover in order to check the colour of a test case.

We call a CSP-CASL refinement '$\leadsto$' *well-behaved*, if, given specifications $(D, P) \leadsto (D', P')$ with consistent data parts D and D', the following holds for any test process T over (D, P):

1. $colour(T) = green$ with respect to (D, P) implies
   $colour(T) = green$ with respect to (D', P'), and

2. $colour(T) = red$ with respect to (D, P) implies
   $colour(T) = red$ with respect to (D', P').

This means: If a test case T reflects a desired behavioural property in (D, P), i.e., $colour(T) = green$, after a well-behaved development step from (D, P) to (D', P') this property remains a desired one and the colour of T is green. If a test case reflects a forbidden behavioural property in (D, P), i.e., $colour(T) = red$, after a well-behaved development step from (D, P) to (D', P') this property remains a forbidden one and the colour of T is red. A well-behaved development step can change only the colour of a test case T involving an open design decision, i.e., $colour(T) = yellow$. For the various CSP-CASL refinements holds: CSP-CASL data refinement is well-behaved; CSP-CASL process refinement based on the CSP models $\mathcal{T}$ is not well-behaved; CSP-CASL process refinement based on CSP models $\mathcal{N}$ and $\mathcal{F}$ is well-behaved, provided the processes involved are divergence-free. [KRS08] studies preservation of test colourings under enhancement.

In terms of refinement, for the above specifications holds: BCALC0 $\rightsquigarrow_{\mathscr{F}}$ BCALC1 $\rightsquigarrow_{\mathscr{F}}$ BCALC2. The first step is a process refinement based on $\mathscr{F}$, the last one is a data refinement, i.e., both refinement steps are well behaved.

In [KRS07, KRS08], we develop these ideas towards a testing framework: Given a test and its colour, its execution on a system under test (SUT) is defined algorithmically. The test result is obtained by comparing intended and actual behaviour of a SUT, where the test verdict is either *pass*, *fail* or *inconclusive*. Intuitively, the verdict *pass* means that the test execution increases our confidence that the SUT is correct with respect to the specification. The verdict *fail* means that the test case exhibits a fault in the SUT, i.e., a violation of the intentions described in the specification. The verdict *inconclusive* means that the test execution neither increases nor destroys our confidence in the correctness of the SUT.

In an industrial cooperation with Rolls-Royce, Derby, UK, our testing approach has successfully been applied to the control software of aeroplane engines, see [HKRS09].

9 Testing framework for EP2

In order to test a concrete EP2 terminal against our CSP-CASL specifications, we built an automatic testing framework Testing EValuator (TEV) [Chu05, Kah10]. TEV is a hardware-in-a-loop on-the-fly testing framework, designed to test an EP2 terminal.

Hardware-in-a-loop testing (HIL) is a well established approach to validate complex systems, where the correct integration of software with its underlying hardware is essential. HIL has been deployed in the defence and aerospace industries as early as the 1950s [NBAR04], nowadays it is an established testing technique. HIL is heavily used in verifying critical system in projects such as the power and thermal control unit of the X-ray satellite "ABRIXAS" [SMH99] and for cabin management controllers for Airbus families [Pel02] and more. Figure 6 illustrates the hardware-in-the-loop testing framework for EP2.

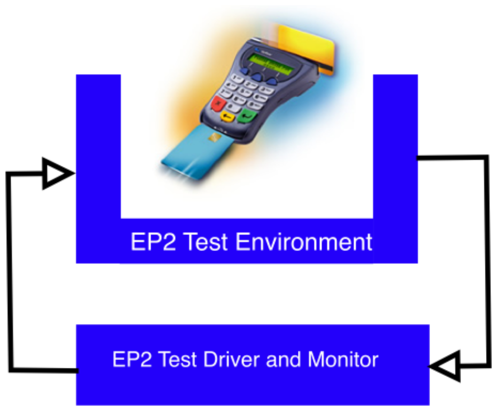

Figure 6: Hardware in the loop testing for EP2.

TEV takes as input a test case protocol, executes the test case and computes the test verdict. The test case protocol contains the following information: the colour of the test case, the EP2 components which are to be tested, timeout information and the actual test case to be executed. In the next section we describe the scenario of a typical test case execution of EP2.

The test verdict is obtained during the execution of the SUT from the expected result defined by the colour of the test process. In [KRS07] we have defined an algorithm which allows to determine the test verdict on the fly. We have implemented this algorithm in TEV, details can be found in [Kah10].

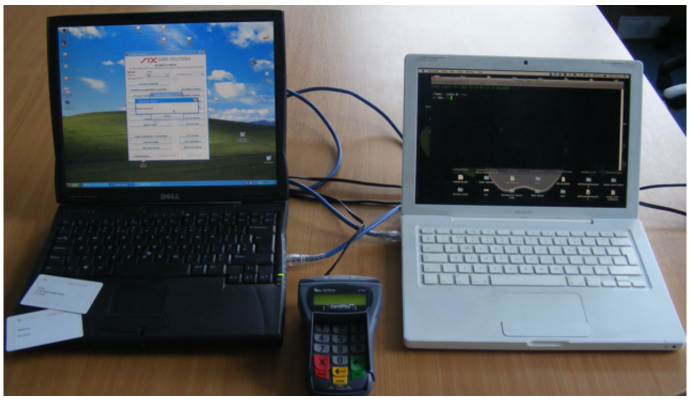
Figure 7: EP2 testing framework in action.

The test execution is conducted in a network environment, where two different machines are connected with a crossed Ethernet cable. Figure 7 illustrates this setting. Here, the EP2 terminal software is running on the black laptop (Windows OS), where a *pinpad* is attached on the serial port. TEV is running on the white laptop (Mac OS).

10 Summary and future work

Figure 8 provides an overview of the CSP-CASL technology discussed: Given an informal system description such as the EP2 documents, we mirror informal specifications by formal ones written in CSP-CASL. In this process, the static analysis of CSP-CASL specifications provides valuable feedback and helps to avoid modelling errors. Then, we analyse if a CSP-CASL specification exhibits desired properties, e.g., deadlock-freedom. On the theoretical side, we showed that – for some fundamental system properties – such an analysis can be carried out by studying an equivalent refinement problem. Thus, on the practical side, CSP-CASL-Prover offers mechanised proof support for the analysis. Additionally, we mirror the informal development steps present in such a system description by CSP-CASL refinement relations. Using CSP-CASL-Prover, we can verify if these refinement relations actually hold. Properties are guaranteed to be preserved within a development step provided a suitable CSP-CASL refinement holds, e.g., deadlock-freedom is preserved under CSP-CASL-stable-failures refinement. Finally, using testing [KRS07, KRS08], we link the formal specifications with actual implementations. CSP-CASL refinement relations

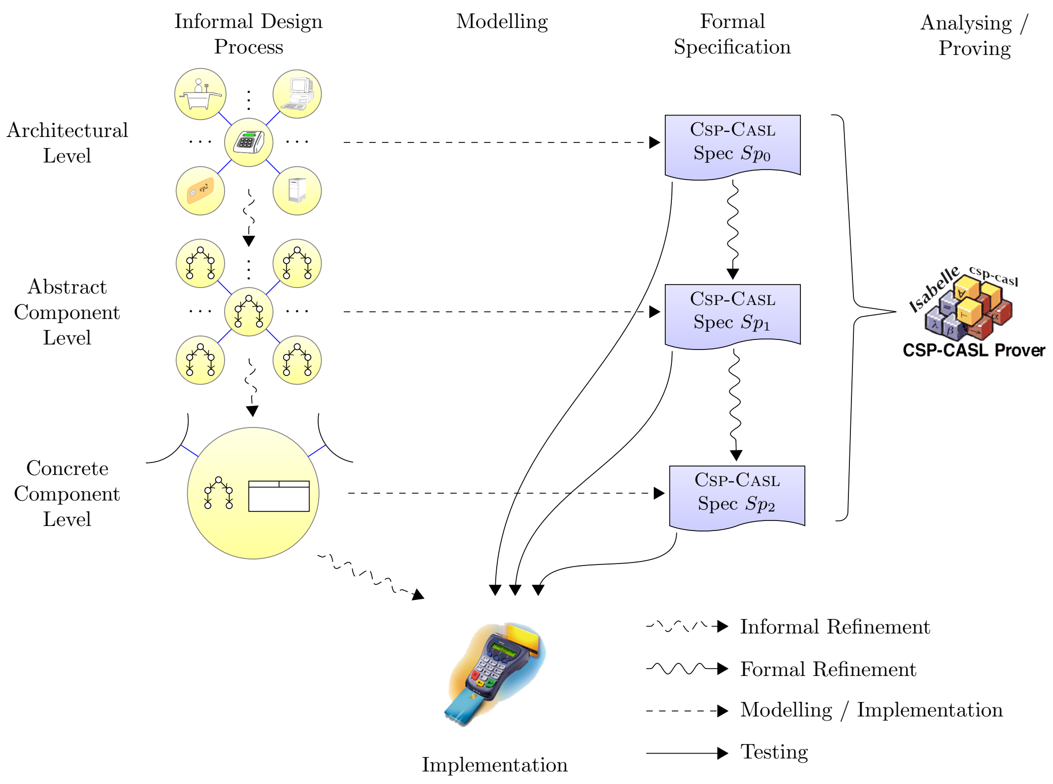

Figure 8: The CSP-CASL verification approach.

ensure that test cases which are designed at an early stage can be used without modification for the test of a later development stage. Hence, test suites can be developed in parallel with the implementation. Also testing has tool support in terms of CSP-CASL-Prover, e.g., in order to colour test cases.

Recently, the CASL institution independent structuring mechanisms have been made available to CSP-CASL, supported by calculi for compositional reasoning on structured CSP-CASL specifications [OMR11a, OMR11b]. It is future work to implement these proof calculi as part of CSP-CASL-Prover and to develop and support a semantically well founded connection from CSP-CASL to distributed programming, e.g., in Java.

Acknowledgement The authors wish to thank B. Krieg-Brückner for inspiring CSP-CASL in the first place, Y. Isobe for his pioneering work on CSP-Prover, T. Mossakowski for advice on institutions and support with HETS, B.-H. Schlingloff for being the driving force in applying CSP-CASL to real world examples, and Erwin R. Catesbeiana (Jr) for being the mastermind of our gang.

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

Bibliography

[ABR99] E. Astesiano, M. Broy, G. Reggio. Algebraic Specification of Concurrent Systems. In Astesiano et al. (eds.), *Algebraic Foundations of Systems Specification*. Springer, 1999.

[CG07] A. L. C. Cavalcanti, M.-C. Gaudel. Testing for Refinement in CSP. In *9th International Conference on Formal Engineering Methods*. LNCS 4789. Springer, 2007.

[CG11] A. L. C. Cavalcanti, M.-C. Gaudel. Testing for Refinement in Circus. *Acta Informatica* 48(2), 2011.

[Chu05] L. B. Chuan. Towards Hardware-In-a-Loop Testing for an International Standard of an Electronic Payment system. Master's thesis, University of Wales Swansea, 2005.

[EP208] EP2 Consortium. EFT/POS 2000 Specification, version 4.0.0. 2008. Project Overview available at http://www.eftpos2000.ch.

[Fis97] C. Fischer. Combining Object-Z and CSP. In Wolisz et al. (eds.), *FBT*. GMD-Studien 315. GMD-Forschungszentrum Informationstechnik GmbH, 1997.

[Gim08] A. Gimblett. Tool Support for CSP-CASL. MPhil thesis, Swansea University. 2008.

[GP95] J. F. Groote, A. Ponse. The syntax and semantics of μCRL. In Ponse et al. (eds.), *Algebra of Communicating Processes '94*. Workshops in Computing. Springer, 1995.

[GRS05] A. Gimblett, M. Roggenbach, B.-H. Schlingloff. Towards a Formal Specification of an Electronic Payment System in CSP-CASL. In Fiadeiro et al. (eds.), *WADT 2004*. LNCS 3423. Springer, 2005.

[HKRS09] G. Holland, T. Kahsai, M. Roggenbach, B.-H. Schlingloff. Towards formal testing of jet engine Rolls-Royce BR725. In Czaja and Szczuka (eds.), *Proc. 18th Int. Conf on Concurrency, Specification and Programming, Krakow, Poland*. 2009.

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

[IR05] Y. Isobe, M. Roggenbach. A Generic Theorem Prover of CSP Refinement. In *TACAS 2005*. LNCS 3440. Springer, 2005.

[IR06] Y. Isobe, M. Roggenbach. A Complete Axiomatic Semantics for the CSP Stable-Failures Model. In Baier and Hermanns (eds.), *CONCUR 2006*. LNCS 4137. Springer, 2006.

[IR07] Y. Isobe, M. Roggenbach. Proof Principles of CSP – CSP-Prover in Practice. In Haasis et al. (eds.), *LDIC 2007*. Springer, 2007.

[IR08] Y. Isobe, M. Roggenbach. CSP-Prover – a Proof Tool for the Verification of Scalable Concurrent Systems. *JCS, JSSST* 25, 2008.

[ISO89] ISO 8807. LOTOS – A formal description technique based on the temporal ordering of observational behaviour. 1989.

[ISO01] ISO/IEC 15437. Information technology – Enhancements to LOTOS (E-LOTOS). 2001.

[Kah87] G. Kahn. Natural Semantics. In *STACS*. 1987.

[Kah10] T. Kahsai. *Property Preserving Development and Testing for CSP-CASL*. PhD thesis, Swansea University, 2010.

[KR09] T. Kahsai, M. Roggenbach. Property preserving refinement notions for CSP-CASL. In Corradini and Montanari (eds.), *WADT 2008*. LNCS 5486. 2009.

[KRS07] T. Kahsai, M. Roggenbach, B.-H. Schlingloff. Specification-based testing for refinement. In Hinchey and Margaria (eds.), *SEFM 2007*. IEEE Computer Society, 2007.

[KRS08] T. Kahsai, M. Roggenbach, B.-H. Schlingloff. Specification-Based Testing for Software Product Lines. In Cerone and Gruner (eds.), *SEFM 2008*. IEEE Computer Society, 2008.

[MML07] T. Mossakowski, C. Maeder, K. Lüttich. The Heterogeneous Tool Set, HETS. In Grumberg and Huth (eds.), *TACAS 2007*. LNCS 4424. Springer, 2007.

[Mos04] P. D. Mosses (ed.). *CASL Reference Manual*. LNCS 2960. Springer, 2004.

[Mos05] T. Mossakowski. Heterogeneous specification and the heterogeneous tool set. Habilitation thesis, Universität Bremen, 2005.

[MRS03] T. Mossakowski, M. Roggenbach, L. Schröder. CoCASL at work – Modelling Process Algebra. In *Coalgebraic Methods in Computer Science*. ENTCS 82. 2003.

[NBAR04] S. Nabi, M. Balike, J. Allen, K. Rzemien. An Overview of Hardware-In-The-Loop Testing Systems at Visteon. *SAE Technical Paper Series*, 2004.

[NPW02] T. Nipkow, L. C. Paulson, M. Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*. LNCS 2283. Springer, 2002.

[OIR09] L. O'Reilly, Y. Isobe, M. Roggenbach. CSP-CASL-Prover – A generic tool for process and data refinement. *ENTCS* 250(2), 2009.

[OMR11a] L. O'Reilly, T. Mossakowski, M. Roggenbach. Compositional Modelling and Reasoning in an Institution for Processes and Data. In *WADT 2010*. LNCS. Springer, 2011. To appear.

[OMR11b] L. O'Reilly, T. Mossakowski, M. Roggenbach. Compositional Reasoning for Processes and Data. In *ARW 2011*. TR-2011-327. Department of Computing Science, University of Glasgow, 2011.

[O'R08] L. O'Reilly. Integrating Theorem Proving for Processes and Data. MPhil thesis, Swansea University. 2008.

[Pel02] J. Peleska. Hardware/Software Integration Testing for the new Airbus Aircraft Families. In Schieferdecker et al. (eds.), *TestCom 2002*. Kluwer, 2002.

[RAC00] G. Reggio, E. Astesiano, C. Choppy. CASL-LTL – A CASL extension for dynamic Reactive Systems. Technical report DISI-TR-99-34, Università di Genova, 2000.

[Rog03] M. Roggenbach. CSP-CASL – A new Integration of Process Algebra and Algebraic Specification. In F.Spoto (ed.), *AMiLP 2003*. TWLT 21. Universiteit Twente, 2003.

[Rog06] M. Roggenbach. CSP-CASL – A new Integration of Process Algebra and Algebraic Specification. *TCS* 354, 2006.

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

[RR00] G. Reggio, L. Repetto. CASL-Charts: a Combination of Statecharts and of the Algebraic Specification Language CASL. In *AMAST 2000*. LNCS 1816. Springer, 2000.

[SAA01] G. Salaün, M. Allemand, C. Attiogbé. A Formalism Combining CCS and CASL. Technical report 00.14, University of Nantes, 2001.

[SMH99] B.-H. Schlingloff, O. Meyer, T. Hülsing. Correctness analysis of an embedded controller. In *DASIA 1999*. ESA SP-447. 1999.

[WC01] J. Woodcock, A. Cavalcanti. A Concurrent Language for Refinement. In Butterfield et al. (eds.), *IWFM 2001*. BCS, 2001.

[WC02] J. Woodcock, A. Cavalcanti. The Semantics of Circus. In Bert et al. (eds.), *ZB 2002*. LNCS 2272. Springer, 2002.