Combining Event-B and CSP: An Institution Theoretic approach to Interoperability

Marie Farrell[*], Rosemary Monahan, and James F. Power

Department of Computer Science, Maynooth University, Ireland

Abstract. In this paper we present a formal framework designed to facilitate interoperability between the Event-B specification language and the process algebra CSP. Our previous work used the theory of institutions to provide a mathematically sound framework for Event-B, and this enables interoperability with CSP, which has already been incorporated into the institutional framework. This paper outlines a comorphism relationship between the institutions for Event-B and CSP, leveraging existing tool-chains to facilitate verification. We compare our work to the combined formalism Event-B||CSP and use a supporting example to illustrate the benefits of our approach.

1 Introduction

Event-B is an industrial strength formal method that allows us to model a system's specification at various levels of abstraction using refinement and prove its safety properties [1]. The most primitive components of an Event-B specification are events, which are triggered non-deterministically once their guards evaluate to true. Much work has been done on imposing control on when events are triggered, as this models state changes in the system [18, 7, 21]. Our contributions seek to provide a mathematical grounding to this work using the theory of institutions and its underlying category theoretic framework [5]. As a result, we provide developers with the ability to add (CSP) control to Event-B specifications. This is achieved through our description of an institution comorphism between an institutional representation of Event-B ($\mathcal{EVT}\mathcal{CASL}$) and an institutional representation of CSP-CASL ($\mathcal{CSP}\mathcal{CASL}$) [16].

This document is structured as follows. In the remainder of section 1 we outline the relevant background, motivate our work, and introduce our running example of a bounded retransmission protocol. Section 2 contains a brief overview of the institutions for $\mathcal{CASL}$ (the Common Algebraic Specification Language), $\mathcal{EVT}\mathcal{CASL}$ and $\mathcal{CSP}\mathcal{CASL}$. In section 3 we outline the comorphism relating the institutions $\mathcal{EVT}\mathcal{CASL}$ and $\mathcal{CSP}\mathcal{CASL}$. We illustrate the use of the syntactic components of this comorphism with respect to our running example in section 4 and discuss implications for refinement of specifications [1, 19]. Finally, we conclude by outlining directions for future work.

---

[*] This work is funded by a Government of Ireland Postgraduate Grant from the Irish Research Council.

```
1  CONTEXT brp_c0
2  SETS STATUS
3  CONSTANTS working, success, failure
4  AXIOMS
5     axm1: STATUS = {working, success,
                      failure}
6     axm2: working ≠ success
7     axm3: working ≠ failure
8     axm4: success ≠ failure
9  END
```

```
10 MACHINE b_0 SEES brp_c0
11 VARIABLES  r_st, s_st
12 INVARIANTS
13    inv1: r_st ∈ STATUS
14    inv2: s_st ∈ STATUS
15 EVENTS
16 Initialisation
17   then
18      act1: r_st := working
19      act2: s_st := working
20 Event brp ≙ ordinary
21   when
22      grd1: r_st ≠ working
23      grd2: s_st ≠ working
24   then
25     Skip
26 Event RCV_progress ≙ anticipated
27   then
28      act1: r_st :∈ {success, failure}
29 Event SND_progress ≙ anticipated
30   then
31      act1: s_st :∈ {success, failure}
32 END
```

Fig. 1: An Event-B model of the bounded retransmission protocol, consisting of a context (lines 1–9) that specifies a new data type called `STATUS`, and a specification for an abstract machine `b_0` (lines 10–32) [1].

1.1 Event-B and a Running Example

Event-B is a state-based formalism for system-level modelling and analysis. It uses set theory as a modelling notation, refinement to represent systems at different levels of abstraction and mathematical proof to verify consistency between refinement levels [1]. In an Event-B model, static aspects of a system are specified in *contexts*, while dynamic aspects are modelled in *machines*. Each machine specifies states and events which update that state. Refinement between machines involves the addition of new variables and events, making the initial model more concrete. Refinement steps generate proof obligations so as to ensure that the refined machine does not invalidate the original model. Event-B is supported by its Eclipse-based IDE, the *Rodin Platform*, which provides support for refinement and automated proof [2].

Figure 1 contains an Event-B specification of a *bounded retransmission protocol* which we use as a running example throughout this paper [1, 19]. The specification corresponds to the sequential file transfer from a sender site to a receiver site [1, Ch. 6]. The Event-B context specifies a data type called `STATUS` (line 2) that contains the three distinct values `working`, `success` and `failure` (lines 3–8). The corresponding abstract machine introduces two state variables of type `STATUS`: these are `r_st` for the receiver and `s_st` for the sender (lines 11–14). The `Initialisation` event (lines 16–19) sets both of these variables to the value `working`.

The events `RCV_progress` and `SND_progress` update the associated state variable to either `success` or `failure` (lines 26–28 and 29–31 respectively). Both events have the status `anticipated` which means that they must not increase the

```
P0 = S0 || R0
S0 = SND_progress → brp → STOP
R0 = RCV_progress → brp → STOP
```

Fig. 2: An Event-B||CSP process specification [19].

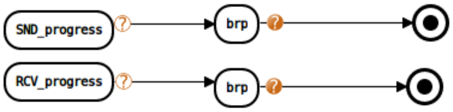

Fig. 3: Using the flows plugin to model the Event-B||CSP process specification in Figure 2.

variant expression in the machine. However, since there is no variant expression in this machine, this condition is not evaluated. While this labelling may seem redundant, it is a common development strategy used in Event-B and, in this case, reminds developers that these events should be refined to `convergent` events in future refinement steps. The event `brp` (lines 20–25) is triggered when both variables are no longer set to `working`, thus indicating that the protocol has completed [19].

1.2 Related work on adding event ordering to Event-B Machines

Developers often wish to model the *order* in which events are triggered, and specifically, how newly added events relate to previous events. Currently, control can only be implemented in Event-B in an ad hoc manner, typically by adding a machine variable to represent the current state. Each event must then check the value of this variable in its guard, and if this value indicates that the machine is ready to move into the next state then the appropriate event is triggered.

An alternative approach to introducing control is provided by the Event-B||CSP formalism which combines Event-B with CSP, so that CSP controllers can be specified alongside Event-B machines facilitating an explicit approach to control flow [18]. CSP is a process algebra specifically designed to specify control oriented applications, using *processes* that can be composed in a variety of ways [6]. The subset of CSP made available by Event-B||CSP is:

$$P ::= e \rightarrow P \;\mid\; P_1 \Box P_2 \;\mid\; P_1 \sqcap P_2 \;\mid\; P_1 \| P_2 \;\mid\; S$$

where P, P_1 and P_2 are processes, e is a CSP event and S is a process variable. The semantics of CSP can be evaluated over a number of semantic domains. These include the traces (sequences of events that a process can engage in after the `Initialisation` event), failures (events the process might refuse after a trace) and divergences (traces after which the process might diverge).

The combination of Event-B and CSP in Event-B||CSP results in a clear separation between the data-dependent and control-dependent aspects of a model, allowing proof obligations concerning control-flow to be discharged within the CSP framework. However, at the time of writing, no tool support has been explicitly provided for this approach, at either the Event-B or CSP level. The ProB animator and model checker can be used to explore Event-B models with CSP controllers for consistency [10]. Since it was not developed with Event-B||CSP in mind there are some incompatibility issues: in particular, it is only feasible to check refinement for small examples.

Figure 2 contains an Event-B||CSP process specification to be used alongside the Event-B model in Figure 1. Here, three CSP processes are defined for use with the machine `b_0`, splitting the specification into sender and receiver controllers ($\mathtt{S}_0$ and $\mathtt{R}_0$ respectively) that are combined in parallel by $\mathtt{P}_0$. This approach was taken by Schneider et al. to model the repeating behaviour of the sender and receiver using CSP, and to model the state using Event-B [19].

Another perspective is provided by the *Flows* plugin for Rodin, which extends Event-B models with event ordering(s) [7]. Flow diagrams represent the possible use cases of Event-B models. These *flows* resemble those used in process algebras such as CSP. A simple graphical notation is used, with a trace semantics provided over the sequence of events in the machine. No new Event-B specifications are generated by the *Flows* plugin. Instead new proof obligations are created to assist reasoning about whether or not a flow is feasible in a given Event-B model. The generated proof obligations characterise the relationship between the after-state of one event and the guard (before-state) of another.

Figure 3 illustrates a potential use case using the *flows* plugin, corresponding to the Event-B||CSP specification in Figure 2, introducing control to the Event-B machine `b_0` (Figure 1). Notice that it is not possible to indicate parallel composition here using the *flows* plugin. We can only specify $\mathtt{S}_0$ and $\mathtt{R}_0$ separately. Therefore, we conclude that the Event-B||CSP specification outlined in Figure 2 is much more expressive that the flow described in Figure 3.

2 Background on Institutions

The theory of institutions was originally developed by Goguen and Burstall in a series of papers originating from their work on algebraic specification [5]. An institution is composed of signatures (vocabulary), sentences (syntax), models and a satisfaction condition (semantics). Figure 4 contains a summary of the definitions for these components. The key observation is that once the syntax and semantics of a formal system have been defined in a uniform way, using some basic constructs from category theory, then a set of *specification-building operators* can be defined that allow specifications to be written and modularised in a formalism-independent manner [17].

Institutions have been defined for many logics and formalisms, including formal languages such as Event-B, UML and CSP [3, 9, 12]. We can achieve interoperability between different logics by constructing a *comorphism* between their institutions. Figure 5 contains a summary of the definitions for the components of an institution comorphism, which broadly consist of mappings for each of the elements in an institution, as referred to in Figure 4. Figures 4 and 5 are brief summaries of the relevant constructions; full details can be found in the literature [5, 17]. Readers familiar with *Unifying Theories of Programming* may note that the notion of institutions, in this way, is similar to that of a "theory supermarket" where one can shop for theories with the confidence that they will work together [4].

An institution is composed of:

Vocabulary: A category **Sign** of *signatures*, with signature morphisms $\sigma : \Sigma \to \Sigma'$ for each signature Σ, $\Sigma' \in |\mathbf{Sign}|$.

Syntax: A functor **Sen** : **Sign** $\to$ *Set* giving a set $\mathbf{Sen}(\Sigma)$ of Σ-*sentences* for each signature Σ and a function $\mathbf{Sen}(\sigma) : \mathbf{Sen}(\Sigma) \to \mathbf{Sen}(\Sigma')$ which translates Σ-sentences to Σ'-sentences for each signature morphism $\sigma : \Sigma \to \Sigma'$.

Semantics: A functor **Mod** : $\mathbf{Sign}^{op} \to \mathbf{Cat}$ giving a category $\mathbf{Mod}(\Sigma)$ of Σ-*models* for each signature Σ and a functor $\mathbf{Mod}(\sigma) : \mathbf{Mod}(\Sigma') \to \mathbf{Mod}(\Sigma)$ which translates Σ'-models to Σ-models (and Σ'-morphisms to Σ-morphisms) for each signature morphism $\sigma : \Sigma \to \Sigma'$.

A Satisfaction Relation $\models_{\mathcal{INS},\Sigma} \subseteq |\mathbf{Mod}(\Sigma)| \times \mathbf{Sen}(\Sigma)$, determining satisfaction of Σ-sentences by Σ-models for each signature Σ.

An institution must uphold the **satisfaction condition:** for any signature morphism $\sigma : \Sigma \to \Sigma'$ the translations $\mathbf{Mod}(\sigma)$ of models and $\mathbf{Sen}(\sigma)$ of sentences

$$M' \models_{\mathcal{INS},\Sigma'} \mathbf{Sen}(\sigma)(\phi) \quad \Leftrightarrow \quad \mathbf{Mod}(\sigma)(M') \models_{\mathcal{INS},\Sigma} \phi$$

for any $\phi \in \mathbf{Sen}(\Sigma)$ and $M' \in |\mathbf{Mod}(\Sigma')|$ [5].

Fig. 4: A brief summary of the definitions for the main components of an institution.

The institutions relevant to this paper are the institutions for CASL, $\mathcal{CASL}$, CSP-CASL, $\mathcal{CSPCASL}$, and our definition of the institution for Event-B, $\mathcal{EVT}$-$\mathcal{CASL}$. Originally, we defined the institution $\mathcal{EVT}$ for Event-B to be built on top of the institution for first-order predicate logic with equality [3]. In this paper, we build our institution $\mathcal{EVTCASL}$ on top of the (more general) institution for $\mathcal{CASL}$, of which $\mathcal{FOPEQ}$ is a sublogic. The main components of these are summarised in Figure 6. We do not delve deeply into their components here, but refer the reader to the literature and our website for further information[1].

The $\mathcal{CSPCASL}$ institution is built on the definition of the institutions $\mathcal{CSP}$ and $\mathcal{CASL}$ [12, 13]. A specification over $\mathcal{CSPCASL}$ consists of a data part (written as a structured CASL specification), a channel part and a process part (written using CSP) [16]. The inclusion of channels is a form of syntactic sugaring as specifications with channels can easily be translated into those without but they provide a more convenient notation so we include them to aid in readability [14].

In Section 3, we outline the institution comorphism between $\mathcal{CSPCASL}$ and our institution for Event-B, $\mathcal{EVTCASL}$. This is the theoretical foundation and main contribution of our work and we use it to create a sound mechanism that has enabled us to achieve interoperability between CSP and Event-B.

2.1 Tool Support and Avenues to Interoperability

The Heterogeneous Toolset (HETS), written in Haskell, provides a general framework for parsing, static analysis and for proving the correctness of specifications in a formalism independent and thus heterogeneous manner [11]. In HETS, each formalism (expressed as an institution) is represented as a logic. In this setting,

---

[1] http://www.cs.nuim.ie/~mfarrell/extended.pdf

An institution comorphism $\rho : \mathbf{INS} \to \mathbf{INS}'$ is composed of:

A functor: $\rho^{Sign} : \mathbf{Sign} \to \mathbf{Sign}'$

A natural transformation: $\rho^{Sen} : \mathbf{Sen} \to \rho^{Sign} ; \mathbf{Sen}'$, that is, for each $\Sigma \in |\mathbf{Sign}|$, a function $\rho^{Sen}_{\Sigma} : \mathbf{Sen}(\Sigma) \to \mathbf{Sen}'(\rho^{Sign}(\Sigma))$.

A natural transformation: $\rho^{Mod} : (\rho^{Sign})^{op} ; \mathbf{Mod}' \to \mathbf{Mod}$, that is, for each $\Sigma \in |\mathbf{Sign}|$, a functor $\rho^{Mod}_{\Sigma} : \mathbf{Mod}'(\rho^{Sign}(\Sigma)) \to \mathbf{Mod}(\Sigma)$.

An institution comorphism must ensure that for any signature $\Sigma \in |\mathbf{Sign}|$, the translations ρ^{Sen}_{Σ} of sentences and ρ^{Mod}_{Σ} of models preserve the satisfaction relation, that is, for any $\psi \in \mathbf{Sen}(\Sigma)$ and $M' \in |\mathbf{Mod}(\rho^{Sign}(\Sigma))|$:

$$\rho^{Mod}_{\Sigma}(M') \models_{\Sigma} \psi \quad \Leftrightarrow \quad M' \models'_{\rho^{Sign}(\Sigma)} \rho^{Sen}_{\Sigma}(\psi)$$

and the relevant diagrams in **Sen** and **Mod** commute for each signature morphism in **Sign** [5].

Fig. 5: A brief summary of the main components of an institution comorphism, which is one way of combining specifications from different institutions.

interoperability between formalisms is defined using institution comorphisms to relate the syntax of different logics and formalisms.

The institutions for $\mathcal{CASL}$ and $\mathcal{CSPCASL}$ have already been implemented in Hets. One notable feature available via Hets is the *CSPCASLProver*, a prover for $\mathcal{CSPCASL}$ based on the CSP-Prover [8]. It uses the Isabelle theorem prover to prove properties about specifications over the permitted CSP semantic domains [15]. We have added an implementation for our institution for Event-B, $\mathcal{EVTCASL}$, to Hets.

In previous work, we have defined a translational semantics for Event-B specifications using the institutional language of $\mathcal{EVTCASL}$. We have implemented this via a parser for the Event-B files that are generated by Rodin. In this way we bridge the gap between the Rodin and Hets software ecosystems, enabling the analysis and manipulation of Event-B specifications in the interoperability-friendly environment made available by Hets. Using our translational semantics for Event-B [3] we generate the $\mathcal{EVTCASL}$ signatures and sentences (as shown in Figure 7) that correspond to the Event-B model defined in Figure 1.

3 Translating $\mathcal{EVTCASL}$ specifications to $\mathcal{CSPCASL}$ specifications

We outline a comorphism-based translation between $\mathcal{EVTCASL}$ and $\mathcal{CSPCASL}$. Both of these institutions rely on $\mathcal{CASL}$ to model the static components of a specification, with Event-B events and CSP processes used to model dynamic behaviour. There are a number of potential approaches to the construction of a comorphism. We could have opted to translate specifications written over both institutions into specifications written over $\mathcal{CASL}$, as $\mathcal{CASL}$ is the base layer of both $\mathcal{EVTCASL}$ and $\mathcal{CSPCASL}$. However, this would lead to the loss of event, channel and process names, unless we used additional annotations alongside the translation. Instead, our approach translates directly from $\mathcal{EVTCASL}$ to

$\mathcal{CASL}$: The institution for CASL [13]:

- **Signatures** are triples of the form $\langle S, \Omega, \Pi \rangle$, containing sort names, sort/arity indexed operation names (representing total and partial functions), sort-indexed predicate names and a subsort relation.
- **Sentences** are first order formulae and term equalities.
- **Models** contain a carrier set corresponding to each sort name, a function over sort carrier sets for each operation name and a relation over sort carrier sets for each predicate name.
- **The satisfaction relation** is the usual satisfaction of first-order structures in first-order sentences.

$\mathcal{CSPCASL}$: The institution for CSP-CASL [16]:

- **Signatures** are tuples $\langle \Sigma_{Data}, C, \Sigma_{Proc} \rangle$ where Σ_{Data} is a basic $\mathcal{CASL}$ signature, C is a set of sort-indexed channel names and $\Sigma_{Proc} = N_{w,comms}$ is a family of finite sets of process names. For every $n \in N_{w,comms}$, w is a sequence of sort names corresponding to the parameter type of n and $comms \subseteq S$ is the set of all types of events that n can engage in.
- **Sentences** are either $\mathcal{CASL}$ sentences or $\mathcal{CSP}$ process sentences.
- **Models** are pairs of the form $\langle A, I \rangle$ where A is a $\mathcal{CASL}$-model and I is a family of process interpretation functions. Each process interpretation function takes as arguments a process name and suitable parameters, and returns a $\mathcal{CSP}$ denotation for the appropriate CSP semantic domain (traces/failures/divergences).
- **The satisfaction relation** for process sentences is two-phase: (i) process terms are evaluated in process sentences using the $\mathcal{CASL}$ semantics, thus replacing each term by its valuation; (ii) the CSP semantics is than applied in the usual way for the specific semantic domain (traces/failures/divergences).

$\mathcal{EVTCASL}$: The institution for Event-B [3]:

- **Signatures** are tuples of the form $\langle S, \Omega, \Pi, E, V \rangle$, where $\langle S, \Omega, \Pi \rangle$ is a $\mathcal{CASL}$ signature, E is a set of (event name, status) pairs, and V is a set of sort-indexed variable names.
- **Sentences** are pairs of the form $\langle e, \phi(\overline{x}, \overline{x}') \rangle$ where e is an event name and $\phi(\overline{x}, \overline{x}')$ is a $\mathcal{CASL}$-formula. Here $\overline{x}$ is the set of free variable names from V and $\overline{x}'$ is the same set with each variable name primed.
- **Models** are triples $\langle A, L, R \rangle$ where A is a $\mathcal{CASL}$ model, L contains sets of variable-to-value mappings for each of the primed versions of the variable names in V. R is a set of relations over the before and after variable-to-value mappings for every (non-initial) event name in E.
- **The satisfaction relation** uses a comorphism between $\mathcal{CASL}$ and $\mathcal{EVT}$ to evaluate the satisfaction of $\mathcal{EVTCASL}$ sentences and models over $\mathcal{CASL}$.

Fig. 6: The principal components of the institutions for the common algebraic specification language ($\mathcal{CASL}$), CSP-CASL ($\mathcal{CSPCASL}$) and Event-B ($\mathcal{EVTCASL}$).

```
Σbrp_c0  =  ⟨ S,  Ω,  Π,  E,  V ⟩ where
  S = {STATUS},
  Ω = {working:STATUS, success:STATUS,
        failure:STATUS},
  Π = {}, E = {}, V = {}

Σb_0  =  ⟨ S,  Ω,  Π,  E,  V ⟩ where
  S = {STATUS,BOOL},
  Ω = {working:STATUS, success:STATUS,
        failure:STATUS},
  Π = {},
  E =
    {(brp, Ordinary),
    (RCV_progress, Anticipated),
    (SND_progress, Anticipated),
    (e_init, Ordinary)},
  V = {(r_st:STATUS), (s_st:STATUS)}
```

The sentences in $\mathbf{Sen}(\Sigma_{brp\_c0})$ that correspond to the Event-B context in Figure 1 are:

```
{⟨e_init, STATUS = {working, success,failure}⟩
⟨e_init, working ≠ success⟩
⟨e_init, working ≠ failure⟩
⟨e_init, success ≠ failure⟩}
```

The sentences in $\mathbf{Sen}(\Sigma_{b\_c0})$ that correspond to the Event-B machine in Figure 1 are:

```
{⟨e_init, STATUS = {working, success, failure}⟩
⟨e_init, working ≠ success⟩
⟨e_init, working ≠ failure⟩
⟨e_init, success ≠ failure⟩
⟨e_init, (r_st' = working ∧ s_st' = working)⟩
⟨brp, (r_st ≠ working ∧ s_st ≠ working)⟩
⟨RCV_progress, (r_st :∈ {success, failure})⟩
⟨SND_progress, (s_st :∈ {success, failure})⟩}
```

Fig. 7: The $\mathcal{EVTCASL}$ signatures and sentences generated, using our translational semantics parser, from the Event-B model in Figure 1. We use subscript notation to indicate the origin of each of these signatures and sentences.

$\mathcal{CSPCASL}$, thus ensuring that the event, channel and process names are not lost. We use the event names in $\mathcal{CSPCASL}$ process definitions in order to introduce control over $\mathcal{EVTCASL}$ specifications.

3.1 An Institution Theoretic Translation

Here we outline the process that we used to define our institution theoretic translation $\rho : \mathcal{EVTCASL} \rightarrow \mathcal{CSPCASL}$ and the difficulties that we encountered. There are three components to an institution comorphism but only the first two are required in order to implement a comorphism translation in HETS. These are the signature and sentences translations described below.

Signature translation:

$$\rho^{Sign} : \mathbf{Sign}_{\mathcal{EVTCASL}} \rightarrow \mathbf{Sign}_{\mathcal{CSPCASL}}$$

Given the $\mathcal{EVTCASL}$ signature $\langle S, \Omega, \Pi, E, V \rangle$, we form the $\mathcal{CSPCASL}$ signature $\langle \Sigma_{Data}, C, \Sigma_{Proc} \rangle$. Since both institutions are based on $\mathcal{CASL}$, we map $\langle S, \Omega, \Pi \rangle$ to Σ_{Data}. We enrich S, the set of sort names, with the new sort `Event` whose carrier set consists of $dom(E)$. For each event name $e \in dom(E)$, we construct the 0-ary operation `e`, of sort `Event`, and add it to Ω. Finally, we equip Σ_{Proc} with the new process names `E_e`, one for each $e \in dom(E)$. Each variable in V is represented by two channels in C of the variable's sort, one for its before value and one for its after value, in order to facilitate variable input for processes.

Sentence translation:

$$\rho^{Sen} : \mathbf{Sen}_{\mathcal{EVTCASL}} \rightarrow \rho^{Sign}; \ \mathbf{Sen}_{\mathcal{CSPCASL}}$$

Each $\mathcal{EVTCASL}$ sentence is of the form $\langle e, \phi(\overline{x}, \overline{x\prime})\rangle$ where e is the event name and $\phi(\overline{x}, \overline{x\prime})$ is a formula over the before and after values of the variables in the signature Σ. As $\mathcal{CSPCASL}$ specifications are over some base logic we assume that this logic corresponds to the base logic of the mathematical predicate language of Event-B for the processes that we construct [12]. Then for each $\mathcal{EVTCASL}$ sentence ρ^{Sen} yields the following $\mathcal{CSPCASL}$ process sentence:

$$\mathtt{E\_e} \quad = \quad ?c_1.\overline{x}_1 \ldots c_{2n}.\overline{x}'_{2n} \rightarrow (\mathtt{if}\ \phi(\overline{x}, \overline{x\prime})\ \mathtt{then}\ \mathtt{e} \rightarrow \mathtt{STOP}\ \mathtt{else}\ \mathtt{STOP})$$

The notation $?c_1.\overline{x}_1 \ldots c_{2n}.\overline{x}'_{2n}$ takes a sort appropriate value for each the variables $\overline{x}_1, \ldots, \overline{x}'_{2n}$ as input on the designated channel for that variable. This indicates that if the formula $\phi(\overline{x}, \overline{x}')$ evaluates to true then the corresponding event e has been triggered. Using the process `STOP` is safe as it does nothing.

Model translation: The signature and sentence translations described above are sufficient for the implementation of an institution comorphism in Hets. However, in order to provide a theoretic underpinning to this translation by correctly defining an institution comorphism we must also provide a translation for the models:

$$\rho^{Mod} : (\rho^{Sign})^{op};\ \mathbf{Mod}_{\mathcal{CSPCASL}} \rightarrow \mathbf{Mod}_{\mathcal{EVTCASL}}$$

Here $\rho^{Mod}(\langle A, I\rangle) = \langle A, L, R\rangle$ and consists of two maps, the identity map on the $\mathcal{CASL}$ model components and a map from I to $\langle L, R\rangle$. Given a $\mathcal{CSPCASL}$-sentence of the form described above, $I(\mathtt{E\_e})$ returns a CSP denotation for the process `E_e` in a specified semantic domain $\mathcal{D} \in \{\mathcal{T}, \mathcal{N}, \mathcal{F}\}$. As the primary concern of Event-B is safety we examine the traces model which gives the following set of traces:

$$\{\ldots, \langle\rangle, \langle c_1.a_1, \ldots, c_{2n}.a_{2n}, \mathtt{e}\rangle, \ldots, \langle c_1.b_1, \ldots, c_{2n}.b_{2n}\rangle, \ldots\}$$

where traces of the form $\langle c_1.a_1, \ldots, c_{2n}.a_{2n}, \mathtt{e}\rangle$ indicate that the predicate $\phi(\overline{x}, \overline{x}')$ evaluated to `true` when the values listed in $c_1.a_1, \ldots, c_{2n}.a_{2n}$ were given to the variables $\overline{x}_1, \ldots, \overline{x}'_{2n}$. Then, traces of the form $\langle c_1.b_1, \ldots, c_{2n}.b_{2n}\rangle$ indicate that the predicate $\phi(\overline{x}, \overline{x}')$ evaluated to `false` on these variable values. We use this traces model to generate the R component (which is made up of the relations $R.e$ for $e \in dom(E) \neq \mathtt{Init}$) of the $\mathcal{EVTCASL}$-model such that:

$$R.e = \{\{\overline{x}_1 \mapsto a_1, \ldots \overline{x}'_{2n} \mapsto a_{2n}\} \mid \langle c_1.a_1, \ldots, c_{2n}.a_{2n}, \mathtt{e}\rangle \in I(\mathtt{E\_e})_{\mathcal{T}}\}$$

Note that in what follows, we abbreviate the `Initialisation` event to `Init`. We only include the values from the traces model that caused the predicate $\phi(\overline{x}, \overline{x}')$ to evaluate to `true`, since these variable values will also satisfy the $\mathcal{EVTCASL}$-sentence $\langle e, \phi(\overline{x}, \overline{x}')\rangle$ in the $\mathcal{EVTCASL}$ institution. These are easily identified as the traces that ended with the event name `e` thus indicating that the event e was triggered. In the case where $\mathtt{e} = \mathtt{Init}$ we construct L in a similar fashion, otherwise, $L = \{\varnothing\}$.

Comorphisms are defined such that for any signature $\Sigma \in |\mathbf{Sign}_{\mathcal{EVTCASL}}|$, the translations $\rho^{Sen}_{\Sigma} : \mathbf{Sen}_{\mathcal{EVTCASL}}(\Sigma) \rightarrow \mathbf{Sen}_{\mathcal{CSPCASL}}(\rho^{Sign}(\Sigma))$ of sentences and $\rho^{Mod}_{\Sigma} : \mathbf{Mod}_{\mathcal{CSPCASL}}(\rho^{Sign}(\Sigma)) \rightarrow \mathbf{Mod}_{\mathcal{EVTCASL}}(\Sigma)$ of models preserve the satisfaction relation. That is, for any $\psi \in \mathbf{Sen}_{\mathcal{EVTCASL}}(\Sigma)$ and $M' \in |\mathbf{Mod}_{\mathcal{CSPCASL}}(\rho^{Sign}(\Sigma))|$

$$\rho^{Mod}_{\Sigma}(M') \models^{\mathcal{EVTCASL}}_{\Sigma} \psi \quad \Leftrightarrow \quad M' \models^{\mathcal{CSPCASL}}_{\rho^{Sign}(\Sigma)} \rho^{Sen}_{\Sigma}(\psi)$$

Note that in the special case where the formula $\phi(\overline{x}, \overline{x}')$ denotes a contradiction (there are no variable values that cause it to evaluate to `true`), then the comorphism satisfaction condition fails to hold. In this case, the corresponding $R.e$ will be empty but as there are variables in the $\mathcal{EVTCASL}$ signature, the generated $\mathcal{EVTCASL}$-model is not a valid one. We are currently investigating alternative constructions of ρ^{Mod} and alternative institution-based translations in order to resolve this issue. The case study that we present in this paper utilises HETS which has no notion of the model translation component of a comorphism so we illustrate how the syntactic components (ρ^{Sign} and ρ^{Sen}) can, in general, be applied to translate $\mathcal{EVTCASL}$ specifications into $\mathcal{CSPCASL}$ specifications that can be processed by HETS.

3.2 Translation via $\boldsymbol{\rho}^{Sign}$ and $\boldsymbol{\rho}^{Sen}$

Figure 8 contains the $\mathcal{CSPCASL}$ specification corresponding to the Event-B specification in Figure 1. Our translation from Event-B to $\mathcal{CSPCASL}$ involves two distinct steps. First, an Event-B specification (Figure 1) is translated into a specification in the language of $\mathcal{EVTCASL}$ using our translational semantics parser (Figure 7). Next, we apply ρ^{Sign} and ρ^{Sen}, the signature and sentence translations described earlier, to the $\mathcal{EVTCASL}$ specification to generate the corresponding $\mathcal{CSPCASL}$ specification (Figure 8). This translation is represented by the dashed arrows in the refinement cube in Figure 10 and the resultant $\mathcal{CSPCASL}$ specification corresponds to the vertex labelled B_0.

Applying ρ^{Sign} to the $\mathcal{EVTCASL}$ signatures in Figure 7 (lines 1–16) generates the $\mathcal{CSPCASL}$ signature $\langle \Sigma_{Data}, C, \Sigma_{Proc} \rangle$ where the sort component of the data signature Σ_{Data} is augmented with new sorts `Event` and `STATUS`. The operation component of Σ_{Data} is augmented with one 0-ary operator per event name in $dom(E)$ of the $\mathcal{EVTCASL}$ signature Σ, yielding the set:

$$\{\texttt{Init}, \texttt{brp}, \texttt{RCV\_progress}, \texttt{SND\_progress} : \texttt{Event}\}$$

C contains two sort-appropriate channels for each variable in V (before and after values). In this $\mathcal{EVTCASL}$ example, there are two variables of sort `STATUS`, yielding four channels of sort `STATUS` in the corresponding $\mathcal{CSPCASL}$ specification. The Σ_{Proc} component of the $\mathcal{CSPCASL}$ signature is augmented a new process `E_e` for every $e \in dom(E)$.

Applying ρ^{Sen} to the sentences in $\mathbf{Sen}(\Sigma)$ (Figure 7, lines 17–28) gives the (syntactically sugared) $\mathcal{CSPCASL}$ specification in Figure 8. Note that we have manually added the process `M` to describe the behaviour of the Event-B machine in its entirety. We use parallel composition to indicate that events are triggered in any order. This specification has been proven consistent, using the Darwin and FACT consistency checkers available in HETS [11]. For readability, we have not included the invariant sentences given in Figure 7 (lines 17–24). The formulae corresponding to each of these sentences is appended by logical conjunction to each of the formulae in the event process definitions in Figure 8 (lines 14–29). We have included the context axiom sentences as predicates (lines 4–8) of the $\mathcal{CSPCASL}$ specification, corresponding to the context in Figure 1.

```
spec BRP_C0 over CASL
  sort STATUS
  ops
  preds STATUS = {working, success,
    failure}
    working ≠ success
    working ≠ failure
    success ≠ failure
end
```

```
spec B_0 over CSPCASL
  data BRP_C0
  channel c1, c2, c3, c4: STATUS
  process
    E_Init =
     ?c1.r_st.c2.s_st.c3.r_st'.c4.s_st' →
       if r_st' = working ∧ s_st' = working
       then (Init → M) else STOP
    E_brp =
     ?c1.r_st.c2.s_st.c3.r_st'.c4.s_st' →
       if r_st ≠ working ∧ s_st ≠ working
       then (brp → M) else STOP
    E_RCV_progress =
     ?c1.r_st.c2.s_st.c3.r_st'.c4.s_st' →
       if r_st :∈ {success, failure}
       then (RCV_progress → M) else STOP
    E_SND_progress =
     ?c1.r_st.c2.s_st.c3.r_st'.c4.s_st' →
       if s_st :∈ {success, failure}
       then (SND_progress → M) else STOP
    M = E_Init ∥ E_brp∥E_RCV_progress
        ∥E_SND_progress
end
```

Fig. 8: *CSPCASL* specification that is generated using ρ^{Sign} and ρ^{Sen} as described in Section 3. This specification has been syntactically sugared for presentation. We provide the full specification that can be input to Hets on our website.

```
1 spec EB||CSP_B_0 over CSPCASL
2   B_0 then
3   process
4     P0 = S0 || R0
5     S0 = SND_progress → brp → STOP
6     R0 = RCV_progress → brp → STOP
7 end
```

```
8 refinement ref0 =
9   B_0 refined to EB||CSP_B_0
```

Fig. 9: A *CSPCASL* specification corresponding to the Event-B‖CSP specification in Figure 2 and a statement of refinement in the notation of Hets between the *CSPCASL* specifications B_0 and EB‖CSP_B_0.

A *CSPCASL* representation of the Event-B‖CSP specification in Figure 2 is illustrated in Figure 9 (lines 1–7). This shows that once the Event-B component of the Event-B‖CSP specification has been translated into *CSPCASL*, then the CSP component can be easily written using *CSPCASL*. These specifications are thus provided with tool support in Hets [11], an environment designed to facilitate interoperability.

4 The Refinement Cube

The refinement cube in Figure 10 depicts the specifications and translations that will be presented throughout this section. In this cube, the labelled vertices represent specifications and the arrows between them describe how they are related. The front face of the cube corresponds to specifications that were developed in Rodin and the combined formalism Event-B‖CSP, the rear face corresponds to those completed in Hets using *CSPCASL*. The vertex labelled `b_0` corresponds

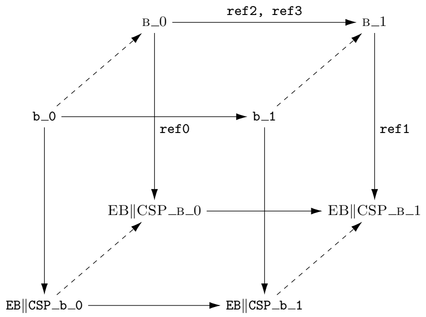

Fig. 10: Refinement cube: solid lines represent refinement relations and the dashed lines represent our translation into $\mathcal{CSPCASL}$.

to the Event-B specification in Figure 1 and the vertex labelled `EB||CSP_b_0` corresponds to the Event-B||CSP specification in Figure 2. The vertical arrow between them indicates that `b_0` is used alongside `EB||CSP_b_0`.

4.1 Event and Process Refinement

In this subsection, we describe the refinement steps that correspond to the solid horizontal arrows in the refinement cube of Figure 10. The theory of institutions equips us with a basic notion of refinement as *model-class inclusion* where the class of models of the concrete specification are a subset of the class of models of the abstract specification [17]. When the signatures are the same we simply denote this refinement as:

$$SP_A \sqsubseteq SP_C \quad \Leftrightarrow \quad Mod(SP_C) \subseteq Mod(SP_A)$$

where SP_A is an abstract specification that refines ($\sqsubseteq$) to a concrete specification SP_C.

If the signatures are different then we must define a signature morphism $\sigma : Sig[SP_A] \to Sig[SP_C]$, and can then use the corresponding model morphism to interpret the concrete specification as containing only the signature items from the abstract specification. This refinement is the model-class inclusion of the models of the concrete specification, restricted using the model morphism, into the class of models of the abstract specification. In this case write:

$$SP_A \sqsubseteq SP_C \quad \Leftrightarrow \quad Mod(\sigma)(SP_C) \subseteq Mod(SP_A)$$

where $Mod(\sigma)(SP_C)$ is the model morphism applied to the model-class of the concrete specification SP_C. This interprets each of the models of SP_C as models of SP_A before a refinement relationship is determined. In our running example, all refinement steps involve a change of signature. A similar approach taken by Schneider et al. involves using a renaming function, f, to relate concrete events to their abstract counterparts before a refinement relation is evaluated [20]. This was used to prove the refinement indicated by the horizontal arrow from `EB||CSP_b_0` to `EB||CSP_b_1` in Figure 10.

Figure 11 contains a refined version of the abstract Event-B machine from Figure 1. Here, each of the events `RCV_progress` and `SND_progress` are refined and split into two events (`RCV_success`, `RCV_failure`, `SND_success` and `SND_failure`). The status of these events has been changed from `anticipated` to `convergent` during the refinement. Thus, the variant expression on line 6 must now be decreased by these events. This amounts to ensuring that in these events the following condition, that we refer to as `var` in Figure 12, holds:

```
MACHINE b_1 refines b_0 SEES brp_c0
VARIABLES r_st, s_st
INVARIANTS
  inv1 s_st = success  ⇒  r_st = success
VARIANT
  {success, failure, s_st, r_st}
Initialisation ordinary
  then
    act1 r_st  :=  working
    act2 s_st  :=  working
Event brp ≙ ordinary
  refines brp
  when
    grd1 r_st  ≠  working
    grd2 s_st  ≠  working
  then
    Skip
Event RCV_success ≙convergent
  refines RCV_progress
  when
    grd1 r_st  =  working
  then
    act1 r_st  :=  success
Event RCV_failure ≙convergent
  refines RCV_progress
  when
    grd1 r_st  =  working
    grd2 s_st  =  failure
  then
    act1 r_st  :=  failure
Event SND_success ≙convergent
  refines SND_progress
  when
    grd1 s_st  =  working
    grd2 r_st  =  success
  then
    act1 s_st  :=  success
Event SND_failure ≙convergent
  refines SND_progress
  when
    grd1 s_st  =  working
  then
    act1 s_st  :=  failure
END
```

Fig. 11: A refined version of the Event-B machine that was described in Figure 1.

$|\{\texttt{success, failure, s\_st', r\_st'}\}| < |\{\texttt{success, failure, s\_st, r\_st}\}|$

When one of the variables moves from `working` to `success` or `failure` then the cardinality of the first set decreases, and this condition will evaluate to true. We apply the same process to this Event-B specification, using our translational semantics and the comorphism that we have described in Section 3. The resulting *CSPCASL* specification is shown in Figure 12.

Specifying refinement between Event-B and Event-B||CSP: We have successfully proven that the Event-B||CSP specification (given in Figure 2 and written as a *CSPCASL* specification in Figure 9 (lines 1–7)) is a refinement of the translation of the Event-B model (given in Figure 1 and written as a *CSPCASL* specification in Figure 8) using the `Auto-DG-Prover` available in Hets.

This refinement is specified in Hets as shown on lines 8–9 of Figure 9, and essentially adds the processes $\mathtt{P}_0$, $\mathtt{S}_0$ and $\mathtt{R}_0$ to the *CSPCASL* specification of `B_0` from lines 10–32 of Figure 8. This inclusion is indicated by the use of the "`then`" specification-building operator (line 2 of Figure 9), which corresponds to proving that the Event-B||CSP specification (Figure 2) is a refinement of the Event-B model (Figure 1). This is a logical conclusion to draw since Event-B||CSP is intended to be used alongside the Event-B machine specification and thus adds a level of deterministic behaviour to the Event-B model.

Similarly, we proved that the Event-B||CSP specification on lines 34–42 of Figure 12 is a refinement of the refined Event-B machine in Figure 11 by translating the Event-B specification into *CSPCASL* via our translational semantics and the comorphism that we outlined earlier. These refinement steps are indicated by the downwards arrows in the back face of the refinement cube in Figure 10 and by the refinement statements `ref0` on lines 8–9 of Figure 9 and `ref1` on line 43 of Figure 12.

```
1  spec B_1 over CSPCASL
2    data BRP_c0
3    channelc1,c2,c3,c4: STATUS
4    process
5      E_Init =
6       ?c1.r_st.c2.s_st.c3.r_st'.c4.s_st' →
7         if r_st' = workings ∧ st' = working
8         then (Init → M) else STOP
9      E_brp =
10      ?c1.r_st.c2.s_st.c3.r_st'.c4.s_st' →
11        if r_st ≠ working ∧ s_st ≠ working
12        then (brp → M) else STOP
13     E_RCV_success =
14      ?c1.r_st.c2.s_st.c3.r_st'.c4.s_st' →
15        if r_st = working ∧ r_st' = success ∧ var
16        then (RCV_success → M) else STOP
17     E_RCV_failure =
18      ?c1.r_st.c2.s_st.c3.r_st'.c4.s_st' →
19        if r_st = working ∧ s_st = failure ∧ var
20          ∧ r_st' = failure
21        then RCV_failure → M) else STOP
22     E_SND_success =
23      ?c1.r_st.c2.s_st.c3.r_st'.c4.s_st' →
24        if s_st = working ∧ r_st = success ∧ var
25          ∧ s_st' = success
26        then (SND_success → M) else STOP
27     E_SND_failure =
28      ?c1.r_st.c2.s_st.c3.r_st'.c4.s_st' →
29        if s_st  = working ∧ s_st' = failure ∧ var
30        then (SND_failure → M) else STOP
31     M = E_Init||E_brp||E_RCV_success||E_SND_success
32         ||E_RCV_failure||E_SND_failure
33 end
```

```
34 spec EB||CSP_B_1 over CSPCASL
35   B_1 then
36   process
37     P_1 = S_1 || R_1
38     S_1 = (SND_success → brp → STOP)
39           □ (SND_failure → brp → STOP)
40     R_1 = (RCV_success → brp → STOP)
41           □ (RCV_failure → brp → STOP)
42 end

43 refinement ref1 = B_1 to EB||CSP_B_1

44 refinement ref2 = B_0 refined via
45   RCV_progress |-> RCV_success
46   SND_progress |-> SND_success
47   E_RCV_progress |-> E_RCV_success
48   E_SND_progress |-> E_SND_success
49 to B_1

50 refinement ref3 = B_0 refined via
51   RCV_progress |-> RCV_failure
52   SND_progress |-> SND_failure
53   E_RCV_progress |-> E_RCV_failure
54   E_SND_progress |-> E_SND_failure
55 to B_1
```

Fig. 12: *CSPCASL* specification corresponding to the concrete machine from Figure 11 as well as the refinement relations. We have also included the corresponding *CSPCASL* for the Event-B||CSP specification used by Schneider et al. on lines (34–42) [19] .

Using CSPCASLProver to preserve Event-B Refinement: Using HETS and the CSPCASLProver we proved a refinement relation between the two *CSPCASL* specifications (Figure 8 and lines 1–33 of Figure 12) that we generated using our comorphism. This is indicated by the top horizontal arrow in the back face of the refinement cube (Figure 10).

Since the corresponding refinement step in Event-B split a single event into two events, we had to define two separate refinements in HETS, `ref2` and `ref3` on lines 44–55 in Figure 12. The syntax of these refinement specifications differs to the previous ones that we have discussed, because this refinement is not the simple addition of processes. Here, the refinement relation specifies the relationship between the signatures of the abstract and refined specifications.

For example, for `ref2` we prove that the following are derivable from the specification in Figure 12:

```
E_RCV_success  =  if r_st' = success ∨ r_st' = failure
                  then RCV_success → M else STOP
E_SND_success  =  if s_st' = success ∨ s_st' = failure
                  then SND_success → M else STOP
            M  =  E_Init||E_brp || E_RCV_success || E_SND_success
```

This corresponds to changing the names of the abstract processes `E_RCV_progress` and `E_SND_progress` to `E_RCV_success` and `E_SND_success` respectively. Thus the concrete processes still preserve the truth of the abstract ones that they refine. A similar construction follows for `ref3`.

Schneider et al. provide a CSP account of Event-B refinement by adding a new event status `devolved`, which indicates events where the CSP controller must ensure convergence [20]. In this paper, we have translated the Event-B specification into $\mathcal{CSPCASL}$ so all convergence checks occur within the same formalism. Therefore we do not need this new status.

These proofs were mostly automatic. Some path issues, caused by the translation from HETS to CSPCASLProver (which uses Isabelle), resulted in a small manual effort to discharge these proofs in Isabelle. Our findings illustrate that the notions of refinement, although expressed differently, in Rodin and HETS are preserved using this comorphism. Thus highlighting the benefits of our institution theoretic approach to interoperability by maintaining that *"truth is invariant under change of notation"* [5].

5 Conclusions and Future Work

Until now, interoperability between Event-B and CSP has been mostly theoretical, offering little in terms of tool support. By devising a means of forming HETS-readable $\mathcal{CSPCASL}$ specifications from those in Event-B we have created tool support for the combination of Event-B and CSP using the theory of institutions. The institutional approach supplies a general framework within which we can achieve interoperability, offering more freedom and a more formal foundation than the approach taken by both the *flows* plugin and the combined formalism Event-B‖CSP, with the advantage of tool support via HETS.

It has been shown that the institutions for both $\mathcal{EVTCASL}$ and $\mathcal{CSPCASL}$ have good behaviour with respect to the institution-theoretic amalgamation property [12, 13]. As a result, we are now able to write *modular* Event-B specifications and interoperate with $\mathcal{CSPCASL}$ using specification-building operators that are made available in the theory of institutions and supported by HETS. In future work, we will investigate the relationships between these specification-building operators and the modularisation constructs in Event-B and CSP. We will define and prove that ρ^{Mod} obeys the required properties. We will also examine whether other kinds of institution morphisms could exist between these two formalisms with particular focus on providing a more heterogeneous specification similar to that of the Event-B‖CSP formalism.

References

1. J.-R. Abrial. *Modeling in Event-B: System and Software Engineering.* Cambridge University Press, New York, NY, USA, 1st edition, 2010.
2. J.-R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *International Journal on Software Tools for Technology Transfer*, 12(6):447–466, 2010.

3. M. Farrell, R. Monahan, and J. F. Power. Providing a Semantics and Modularisation Constructs for Event-B using Institutions. In *International Workshop on Algebraic Development Techniques*, 2016.
4. J. Fitzgerald, P. G. Larsen, and J. Woodcock. Foundations for model-based engineering of systems of systems. In *Complex Systems Design & Management*, pages 1–19. Springer, 2014.
5. J. A. Goguen and R. M. Burstall. Institutions: abstract model theory for specification and programming. *Journal of the ACM*, 39(1):95–146, 1992.
6. C. A. R. Hoare. Communicating sequential processes. In *The origin of concurrent programming*, pages 413–443. Springer, 1978.
7. A. Iliasov. On Event-B and control flow. Technical report, Newcastle University, Newcastle Upon Tyne, U.K., 2009.
8. Y. Isobe and M. Roggenbach. CSP-Prover – a proof tool for the verification of scalable concurrent systems. *Information and Media Technologies*, 5(1):32–39, 2010.
9. A. Knapp, T. Mossakowski, M. Roggenbach, and M. Glauer. An institution for simple UML state machines. In *Fundamental Approaches to Software Engineering*, volume 9033 of *LNCS*, pages 3–18, 2015.
10. M. Leuschel and M. Butler. ProB: an automated analysis toolset for the B method. *International Journal on Software Tools for Technology Transfer*, 10(2):185–203, 2008.
11. T. Mossakowski, C. Maeder, and K. Lüttich. The heterogeneous tool set, HETS. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4424 of *LNCS*, pages 519–522, 2007.
12. T. Mossakowski and M. Roggenbach. Structured CSP – a process algebra as an institution. In *Recent Trends in Algebraic Development Techniques*, volume 4409 of *LNCS*, pages 92–110, 2007.
13. P. D. Mosses, editor. *CASL Reference Manual*, volume 2960 of *LNCS*. Springer, 2004.
14. L. O'Reilly. *Structured Specification with Processes and Data*. PhD thesis, Swansea University, Swansea, U.K., 2012.
15. L. O'Reilly, M. Roggenbach, and Y. Isobe. CSP-CASL-Prover: a generic tool for process and data refinement. *Electronic Notes in Theoretical Computer Science*, 250(2):69–84, 2009.
16. M. Roggenbach. CSP-CASL – a new integration of process algebra and algebraic specification. *Theoretical Computer Science*, 354(1):42–71, 2006.
17. D. Sanella and A. Tarlecki. *Foundations of Algebraic Specification and Formal Software Development*. Springer, 2012.
18. S. Schneider, H. Treharne, and H. Wehrheim. A CSP approach to control in Event-B. In *Integrated Formal Methods*, volume 6396 of *LNCS*, pages 260–274, 2010.
19. S. Schneider, H. Treharne, and H. Wehrheim. Bounded retransmission in Event-B||CSP: a case study. *Electronic Notes in Theoretical Computer Science*, 280:69–80, 2011.
20. S. Schneider, H. Treharne, and H. Wehrheim. The behavioural semantics of Event-B refinement. *Formal Aspects of Computing*, 26:251–280, 2014.
21. C. Snook and M. Butler. UML-B and Event-B: an integration of languages and tools. In *IASTED International Conference on Software Engineering*, pages 336–341, Innsbruck, Austria, 2008.